From Safety To Termination And Back: SMT-Based Verification For Lazy Languages

Niki Vazou  Eric L. Seidel  Ranjit Jhala
UC San Diego

Abstract
SMT-based verifiers have long been an effective means of ensuring safety properties of programs. While these techniques are well understood, we show that they implicitly require eager semantics; directly applying them to a lazy language is unsound due to the presence of divergent sub-computations. We recover soundness by composing the safety analysis with a termination analysis. Of course, termination is itself a challenging problem, but we show how the safety analysis can be used to ensure termination, thereby bootstrapping soundness for the entire system. Thus, while safety invariants have long been required to prove termination, we show how termination proofs can be to soundly establish safety.

1. Introduction
SMT-based verifiers, based on Floyd-Hoare Logic (e.g. EscJava) , or combined with abstract interpretation (e.g. SLAM), have been highly effective at the automated verification of imperative programs. In the functional setting, these techniques are generalized as refinement types, where invariants are encoded by composing types with SMT-decidable refinement predicates. For example

\[\text{type } \text{Pos} = \{v: \text{Int} \mid v > 0\}\]

\[\text{type } \text{Nat} = \{v: \text{Int} \mid v >= 0\}\]

are the basic type \text{Int} refined with logical predicates that state that “the values” \(v\) described by the type are respectively strictly positive and non-negative. We encode pre- and post-conditions (contracts) using refined function types like

\[\text{div} :: \text{n: Nat} \rightarrow \text{d: Pos} \rightarrow \{v: \text{Nat} \mid v <= n\}\]

which states that the function \text{div} requires inputs that are respectively non-negative and positive, and ensures that the output is less than the first input \(n\). If a program containing \text{div} statically type-checks, we can rest assured that executing the program will not lead to any unpleasant divide-by-zero errors. Several groups have demonstrated that refinements can be used to statically verify properties ranging from simple array safety to functional correctness of data structures, security protocols, and compiler correctness.

Given the remarkable effectiveness of the technique, we embarked on the project of developing a refinement type based verifier for Haskell, assuming that the standard soundness proofs from Floyd-Hoare logics and refinement types would carry over directly. Of course, the previous logics and systems were all developed for eager, call-by-value languages, but we presumed that the order of evaluation would surely prove irrelevant, and that the soundness guarantees would translate to Haskell’s lazy, call-by-need regime.

To our surprise, we were totally wrong.

1. Laziness Precludes Partial Correctness
Our first contribution is to demonstrate that refinement typing is unsound in the presence of lazy evaluation. Consider the program:

\[
\begin{align*}
\text{foo} & :\text{ n: Nat} \rightarrow \{v: \text{Nat} \mid v < n\} \\
\text{foo} n &= \begin{cases} 
0 & (n > 0) \\
\text{foo} (n-1) & (n \leq 0)
\end{cases} \\
\text{bar} & : \text{ z: Pos} \rightarrow \text{x: Int} \rightarrow \text{Int} \\
\text{bar} z x &= 2014 \ 'z'
\end{align*}
\]

\[
\begin{align*}
\text{main} &= \text{let } (a,b) = (0, \text{foo} 0) \ \text{in} \ \text{bar} a b
\end{align*}
\]

A standard refinement type checker will happily verify the above program. The refinement type signature for \text{foo} captures the partial correctness property: the function \text{foo} requires non-negative inputs and ensures that its output (if one is produced!) will be strictly less than its input. Consequently, the checker concludes that at the call-site for \text{bar}, the value \(z\) equals 0 and the value \(x\) is some non-negative integer that is strictly less than 0. In other words, the checker concludes that the environment is inconsistent and hence trivially type checks.

The issue is independent of refinement typing and affects any Floyd-Hoare logic-based verifier. For example, a hypothetical ESC-Scala would verify the following code which restates the above using classical requires and ensures clauses:

\[
\begin{align*}
\text{def } \text{foo}(\text{n: Int}): \text{Int} &= \\
&\begin{cases} 
0 & (n <= \text{result} && \text{result < n}) \\
\text{foo} (n-1) & (n > 0)
\end{cases} \\
\text{def } \text{bar}(\text{z: Int}, \text{x: Int}) &= \\
&\begin{cases} 
\text{result} & (0 < \text{z}) \\
\text{return} (2014 \ 'z') & (0 = \text{z})
\end{cases}
\end{align*}
\]

One should not be alarmed as this deduction is perfectly sound under eager, call-by-value semantics. In both cases, the verifier determines that the call to \text{bar} is dead code – the call is safe because it is not invoked at all. This reasoning is quite unsound for Haskell’s lazy, call-by-need semantics, and Scala’s lazy, call-by-name parameters (indicated by the := type annotation). In both cases, the program execution would skip blithely over the call to \text{foo}, plunge headlong into the \text{div}, and crash.

As we show, the problem is that with lazy evaluation, one can only trust a refinement or invariant if one is guaranteed that evaluating the corresponding term will not diverge. That is, the classical...
Floyd-Hoare separation of “partial” and “total” correctness breaks down, and even safety verification requires checking termination.

2. Termination and Refinement Typing The prognosis seems dire: to solve one problem it appears we must first solve a harder one! Our second contribution is to demonstrate that refinement types can themselves be used to prove termination (§2).

In particular, we show how to adapt the classical idea of ranking functions [5], as embodied via sized types [2, 15], to the setting of Refinement Typing. The key idea is to ensure that each recursive call is made with parameters of strictly decreasing Nat-valued size.

We show that refinements naturally encode sized types and generalize them in useful ways by allowing (1) different notions of size to account for recursive data types, (2) lexicographically ordered ranking functions to support complex forms of recursion, and most importantly (3) the use of auxiliary relational invariants (circularly, via refinement types!) to verify that sizes decrease in non-structurally recursive functions.

Our use of refinements to prove termination makes proving soundness interesting in two ways. First, to check Haskell codebases, we cannot require that every term terminates, and so we must support programs containing terms that may diverge. Second, there is a circularity in the soundness proof itself as termination is required to prove refinements sound and vice versa. We address these issues by developing a core calculus (§3) with optimistic semantics ([1]), proving those semantics equivalent to call-by-name evaluation, and then proving soundness with respect to the optimistic semantics (§4). Thus, while it is well known that safety properties (invariants) are needed to prove termination [8], we show for the first time how termination properties are needed to prove safety.

3. Refinement Types for Real-World Haskell We have implemented our technique by extending LIQUIDHASKELL [3]. Our third contribution is an experimental evaluation of the effectiveness of our approach by using LIQUIDHASKELL to check substantial, real-world Haskell libraries totaling over 10,000 lines of code (§5). The verification of these libraries requires precisely analyzing recursive structures like lists and trees, tracking the relationships between their contents, inferring invariants in the presence of low-level pointer arithmetic, and lifting the analysis across polymorphic, higher-order functions. We demonstrate that by using LIQUIDHASKELL we were able to prove termination and a variety of critical safety and functional correctness properties with a modest number of manually specified hints, and even find and fix a subtle correctness bug related to uncode handling in TEXT.

To the best of our knowledge, this is the most substantial evaluation of refinement types on third party code, and demonstrates that SMT-based safety- and termination-verification can be practical and effective for higher-order, functional languages.

2. Overview

We start with an overview of our contributions. After quickly recapitulating the basics of refinement types we illustrate why the approach is unsound in the presence of lazy evaluation. Next, we show that we can recover soundness by coupling refinements with a termination analysis. Fortunately, we demonstrate how we can bootstrap off of refinements to solve the termination problem, and hence obtain a sound and practical verifier for Haskell.

SMT-Based Refinement Type Checking Recall the refinement type aliases Pos and Nat and the specification for div from §1. A refinement type system will use these to reject

```haskell
bad :: Nat -> Nat -> Int
bad x y = x ‘div’ y
```

because, to check that the second parameter y has type Pos at the call to div, the system will issue a subtyping query

\[ x : \{ x \geq 0 \}, \; y : \{ y \geq 0 \} \vdash \{ v = y \} \preceq \{ v > 0 \} \]

which reduces to the invalid SMT query

\[ (x \geq 0) \land (y \geq 0) \Rightarrow (v = y) \Rightarrow (v > 0) \]

On the other hand, the system will accept the program

```haskell
good :: Nat -> Nat -> Int
good x y = x ‘div’ (y + 1)
```

Here, the corresponding subtyping query is

\[ x : \{ x \geq 0 \}, \; y : \{ y \geq 0 \} \vdash \{ v = y + 1 \} \preceq \{ v > 0 \} \]

which reduces to the valid SMT query

\[ (x \geq 0) \land (y \geq 0) \Rightarrow (v = y + 1) \Rightarrow (v > 0) \]

2.1 Laziness Makes Refinement Typing Unsound

Next, let us look in detail at the program with foo and bar from §1. A standard refinement type checker first verifies the signature of foo in the classical rely-guarantee fashion, by (inductively) assuming its type and checking that its body yields the specified output. In the then branch, the output subtyping obligation

\[ n : \{ n \geq 0 \}, \; \vdash \{ v = n - 1 \} \preceq \{ 0 \leq v \land v < n \} \]

reduces to the valid SMT formula

\[ n \geq 0 \land n > 0 \Rightarrow (v = n - 1) \Rightarrow (0 \leq v \land v < n) \]

In the else branch, the output is proven by inductively using the assumed type for foo. Next, inside main the binder b is assigned the output type of foo with the formal n replaced with the actual 0. Thus, the subtyping obligation at the call to bar is

\[ a : \{ a = 0 \}, \; b : \{ 0 \leq b \land b < 0 \} \vdash \{ v = a \} \preceq \{ v > 0 \} \]

which reduces to the SMT query

\[ a = 0 \land (0 \leq b \land b < 0) \Rightarrow (v = a) \Rightarrow (v > 0) \]

which is trivially valid as the antecedent is inconsistent.

Unfortunately, this inconsistency is unsound under Haskell’s lazy evaluation! Since b is not required, the program will dive headlong into evaluating the div and hence crash, rendering the type checker’s guarantee meaningless.

Reconciling Laziness And Refinements One may be tempted to get around the unsoundness via several different routes. First, one may be tempted to point the finger of blame at the “inconsistency” itself. Unfortunately, this would be misguided, since such inconsistencies are not a bug but a crucial feature of refinement type systems. They enable, among other things, path sensitivity by incorporating information from run-time tests (guards) and hence let us verify that expressions that throw catastrophic exceptions (e.g. error e) are indeed unreachable dead code and will not explode at run-time. Second, one might use a CPS transformation [25, 26] to convert the program into call-by-value. We confess to be somewhat wary of the prospect of translating inferred types and errors back to the source level after such a transformation. Previous experience shows that the ability to map types and errors to source is critical for usability. Third, one may want some form of strictness analysis [20] to statically predict which expressions must be evaluated, and only use refinements for those expressions. This route is problematic as it is unclear whether one can develop a sufficiently precise strictness analysis. More importantly, it is often useful to add ghost values into the program for the sole purpose of making refinement
types complete \[33\]. By construction these values are not used by the program, and would be thrown away by a strictness analysis, thus precluding verification.

### 2.2 Ensuring Soundness With Termination

The crux of the problem is that when we establish that

\[
\Gamma \vdash e : \{v : \text{Int} \mid p\}
\]

what we have guaranteed is that if \( e \) reduces to an integer \( n \), then \( n \) satisfies the logical predicate \( p \{n/v\} \) \[14\] \[19\]. Thus, to account for diverging computations, we should properly view the above typing judgment as weakened with a bottom disjunct.

\[
\Gamma \vdash e : \{v : \text{Int} \mid v = \bot \lor p\}
\]

Now, consider the expression \texttt{let } \( x = e \) in \( e' \). In an eager setting, we can readily eliminate the bottom disjunct and assume that \( x \) satisfies \( p \{x/v\} \) when analyzing \( e' \) because if \( e \) diverges, then \( e' \) is not evaluated. In other words, the mere fact that evaluation of \( e' \) began allows us to conclude that \( x \neq \bot \), and so we can eliminate the bottom disjunct without compromising soundness. However, in a lazy setting we cannot drop the bottom disjunct because we may well evaluate \( e' \) even if \( e \) diverges!

One way forward is to take the bull by the horns and directly reason about divergence and laziness using the bottom disjunct. That is, to weaken each refinement with the bottom disjunct. While sound, such a scheme is imprecise as the hypotheses will be too weak to let us prove interesting relational invariants connecting different program variables. For instance, the subtyping query for \texttt{good} \[14\], if we ignore the \( x \) binder, becomes:

\[
y : \{y = \bot \lor y \geq 0\} \vdash \{v = \bot \lor v = y + 1\} \leq \{v = \bot \lor v > 0\}
\]

which boils down to the SMT query

\[
(y = \bot \lor y \geq 0) \Rightarrow (v = \bot \lor v = y + 1) \Rightarrow (v = \bot \lor v > 0)
\]

which is invalid, causing us to reject a perfectly safe program!

One might try to make the direct approach a little less naive by somehow axiomatizing the semantics of operators like \( + \) to stipulate that the result (e.g. \( v \) above) is only non-bottom when the operands (e.g. \( y \) and \( 1 \)) are non-bottom. In essence, such an axiomatization would end up encoding lazy evaluation inside the SMT solver’s logic, and would indeed make the above query valid. However, it is quite unclear to us how to design such an axiomatization in a systematic fashion. Worse, even with such an axiomatization, we would never be able to verify trivial programs like

\[
\text{baz} :: \text{Int} \rightarrow \text{Int} \rightarrow \{z : \text{Int} \mid z > 0\}
\]

\[
\text{baz } x y z = \text{assert} (x > y) 0
\]

as the bottom disjunct on \( z \) – which is never evaluated – would preclude the verifier from using the refinement relating the values of \( x \) and \( y \) that is needed to prove the assertion. The above example is not contrived; it illustrates a common idiom of using ghost variables that carry "proofs" about other program variables.

The logical conclusion of the above line of inquiry is that to restore soundness and preserve precision, we need a means of precisely eliminating the bottom disjunct, i.e. of determining when a term is definitely not going to diverge. That is, we need a termination analysis. With such an analysis, using Reynolds’ \[20\] terminology, we could additionally type each term as either trivial, meaning it \texttt{must} terminate, or serious, meaning it \texttt{may not} terminate. Furnished with this information, the refinement type checker may soundly drop the bottom disjunct for trivial types, and keep the bottom disjunct (or just enforce the refinement \texttt{true}) for serious terms.

Our approach is properly viewed as an optimization that strengthens the “direct” refinement (with a bottom disjunct) with a termination analysis that lets us eliminate the bottom disjunct in the common case of terminating terms. In general, this approach leaves open the possibility of directly reasoning about bottom (e.g. when reasoning about infinite streams), as we do not require that all terms be provably terminating, but only the ones with (non-trivial) refinements \texttt{without} the bottom disjunct. The dual approach – using safety invariants to strengthen and prove termination is classical \[8\]; this is the first time termination has been used to strengthen and prove safety!

As an aside, readers familiar with fully dependently typed languages like Agda \[21\] and Coq \[4\] may be unsurprised at the termination requirement. However, the role that termination plays here is quite different. In those settings, arbitrary terms may appear in types; termination ensures the semantics are well-defined and facilitates type equivalence. In contrast, refinement logics are carefully designed to preclude arbitrary terms; they only allow logical predicates over well-defined, decidable theories, which crucially also include program variables. As we saw, this choice is sound under call-by-value, but problematic under call-by-name, as in the latter setting even a mere first-order variable can correspond to an undefined diverging computation.

### 2.3 Ensuring Termination With Refinements

How shall we prove termination? Fortunately, there is a great deal of research on this problem. The heart of almost all the proposed solutions is the classical notion of ranking functions \[32\]: in any potentially \texttt{looping} computation, prove that some well-founded metric strictly decreases every time around the loop.

#### Sized Types

In the context of typed functional languages, the primary source of looping computations is recursive functions. Thus, the above principle can be formalized by associating a notion of size with each type, and verifying that in each recursive call, the function is invoked with arguments whose size is strictly smaller than the current inputs. Thus, one route to recovering soundness would be to perform a first phase of size analysis like \[2,13,24\] to verify the termination of recursive functions, and then carry out refinement typing in a second phase. However, (as confirmed by our evaluation) proving that sizes decrease often requires auxiliary invariants of the kind refinements are supposed to establish in the first place \[8\]. Instead, like \[33\], we develop a means of encoding sizes and proving termination circularly, via refinement types.

**Using The Value As The Size** Consider the function

\[
\text{fib} :: \text{Nat} \rightarrow \text{Int}
\]

\[
\text{fib } 0 = 1
\]

\[
\text{fib } 1 = 1
\]

\[
\text{fib } n = \text{fib } (n-1) + \text{fib } (n-2)
\]

Recall that in our setting \texttt{Nat} is simply non-negative \texttt{Integer}. Thus, we can naturally associate a size with its own value.

**Termination via Environment-Weakening** To verify safety, a standard refinement type checker (or Houare-logic based program verifier) would check the body assuming an environment:

\[
\text{n : Nat, fib : Nat } \rightarrow \text{Int}
\]

Consequently, it would verify that at the recursive call-site the argument \( n - 2 \) is a \texttt{Nat}, by checking the SMT validity of

\[
(n \geq 0 \land n \neq 0 \land n \neq 1) \Rightarrow (v = n - 2) \Rightarrow v \geq 0
\]
and the corresponding formula for \( n - 1 \), thereby guaranteeing that \( \text{fib} \) respects its signature. For termination, we tweak the procedure by checking the body in a termination-weakened environment

\[
n : \mathbb{N}, \text{fib} : \{ n' : \mathbb{N} | n' < n \} \rightarrow \mathbb{N}
\]

where we have weakened the type of \( \text{fib} \) by stipulating that \( \text{it only be recursively called with } \mathbb{N} \) values \( n' \) that are strictly less than the current parameter \( n \). The body still type checks as

\[
(n \geq 0 \land n \neq 0 \land n \neq 1) \Rightarrow (v = n - 2) \Rightarrow v \geq 0 \land v < n
\]

is a valid SMT formula. We prove (Theorem 3) that since the body typechecks under the weakened assumption for the recursive binder, the function will terminate on all \( \mathbb{N} \).

**Lexicographic Termination** Our environment-weakening technique generalizes to include so-called lexicographically decreasing measures. For example, consider the Ackermann function.

\[
\text{ack} :: \mathbb{N} \rightarrow \mathbb{N} \rightarrow \mathbb{N}
\]

\[
\text{ack} \ m \ n = \begin{cases} 
 0 & \text{if } n = 0 \\
 1 & \text{if } m = 0 \\
 \text{ack} \ (m-1) \ 1 & \text{otherwise}
\end{cases}
\]

We cannot prove that some argument always decreases; the function terminates because either the first argument strictly decreases or the first argument remains the same and the second argument strictly decreases, i.e., the pair of arguments strictly decreases according to a well-founded lexicographic ordering. To account for such functions, we generalize the notion of weakening to allow a sequence of witness arguments while requiring that (1) each witness argument be non-increasing, and (2) the last witness argument be strictly decreasing if the previous arguments were equal. These requirements can be encoded by generalizing the notion of environment-weakening: we check the body of \( \text{ack} \) under

\[
m : \mathbb{N}, \\
n : \mathbb{N},
\]

\[
\text{ack} : m : \mathbb{N} \rightarrow \{ n' : \mathbb{N} | m' \leq m \lor m' = m \Rightarrow n' < n \} \rightarrow \mathbb{N}
\]

Thus, while sound refinement typing requires proving termination, on the bright side, refinements make proving termination easy.

**Measuring The Size of Structures** Consider the function \( \text{map} \) defined over the standard list type.

\[
\text{map} f \ [] = []
\]

\[
\text{map} f \ ys@(x:xs) = f \ x : \text{map} \ f \ xs
\]

In \( \text{map} \), the recursive call is made to a “smaller” input. We formalize the notion of size with \( \text{measure} \).

\[
\text{measure} \ \text{len} :: [a] \rightarrow \mathbb{N}
\]

\[
\text{len} \ [] = 0
\]

\[
\text{len} \ (x:xs') = 1 + (\text{len} \ xs')
\]

With the above definition, the \( \text{measure} \) strengthens the type of the data constructors to:

\[
[] :: (v : [a] | \text{len} \ v = 0)
\]

\[
(\cdot) :: x : a \rightarrow x : [a]
\]

\[
\rightarrow (v : [a] | \text{len} \ v = 1 + \text{len} \ xs)
\]

where \( \text{len} \) is simply an uninterpreted function in SMT logic \( [18] \).

We can now verify that \( \text{map} \) does not change the length of the list:

\[
\text{type} \ \text{ListEq} \ a \ YS = (v : [a] | \text{len} \ v = \text{len} \ YS)
\]

\[
\text{map} :: (a \rightarrow b) \rightarrow ys : [a] \rightarrow \text{ListEq} \ b \ ys
\]

This type is only valid if \( \text{map} \) provably terminates. We simultaneously verify termination and the type as before, by checking the body in the termination-weakened environment

\[
y s : [a]
\]

\[
\text{map} : (a \rightarrow b) \rightarrow (ys' : [a] | \text{len} \ ys' < \text{len} \ ys) \rightarrow \text{ListEq} \ b \ ys'
\]

To ensure that the body recursively uses \( \text{map} \) per its weakened specification, the recursive call \( \text{map} \ f \ xs \) generates the subtyping query

\[
y s : (\text{len} \ ys = 1 + \text{len} \ xs)
\]

\[
xs : (\text{len} \ xs \geq 0) \vdash \{ y s' = x s \} \preceq \{ \text{len} \ ys' < \text{len} \ ys\}
\]

Thanks to the case unfolding, the type of \( ys \) is strengthened with the measure relationship with the tail \( xs \) \( [18] \). Hence, the subtyping above reduces to the valid SMT query

\[
\text{len} \ ys = 1 + \text{len} \ xs
\]

\[
\land \text{len} \ xs \geq 0 \Rightarrow (ys' = x s) \Rightarrow (\text{len} \ ys' < \text{len} \ ys)
\]

Hence, using refinements and environment-weakening, we simultaneously verify termination and the output type for \( \text{map} \).

**Witnessing Termination** Sometimes, the decreasing metric cannot be associated with a single parameter or lexicographically ordered sequence of parameters, but is instead an auxiliary value that is a function of the parameters. For example, here is the standard merge function from the eponymous sorting procedure:

\[
\text{merge} xs@(x:xs') \ ys@(y:ys') = \begin{cases} 
 x < y & \Rightarrow x : \text{merge} xs' \ ys \\
 x \leq y & \Rightarrow y : \text{merge} dx \ ys \ ys'
\end{cases}
\]

where

\[
dx = \text{length} \ xs + \text{length} \ ys
\]

\[
dy = \text{length} \ dx + \text{length} \ ys'
\]

where \( \text{length} :: zs : [a] \rightarrow (v : \mathbb{N} | v = \text{len} \ zs) \) returns the number of elements in the list. Now, the system verifies that \( d \) equals the sum of the sizes of the two lists, and that in each recursive call \( dx \) and \( dy \) are strictly smaller than \( d \), thereby proving that \( \text{merge} \) terminates.

3. Language

Next, we present a core calculus \( \lambda_t \) that formalizes our approach of refinement types under lazy evaluation. Instead of proving soundness directly on lazy, call-by-name (CBN) semantics, we prove soundness with respect to an optimistic (OPT) semantics where (provably) terminating terms are eagerly evaluated, and we separately prove an equivalence relating the CBN and OPT evaluation strategies. With this in mind, let us see the syntax (§3.1), the dynamic semantics (§3.2), and finally, the static semantics (§3.3).

3.1 Syntax

Figure 1 summarizes the syntax of expressions and types.

**Constants** The primitive constants include basic values like \( \text{true}, \text{false}, 0, 1, \ etc. \), arithmetic and logical operators like \( +,-,\leq,\land,\neg \). In addition, we include a special (untypable) \( \text{crash} \) constant that models errors. Primitive operations will return a \( \text{crash} \) when
Forcing Evaluation

A small-step contextual operational semantics is defined in our rules, contexts, and values as defined as usual. In addition to the usual lazy semantics, our soundness proof requires a “helper” optimistic semantics. Thus, we parameterize the small step rules with a force predicate \( e \in \text{Fin} \) that is used to determine whether to force evaluation of the expression \( e \) or to defer evaluation until the value is needed.

Evaluation Rules

We have structured the small step rules so that if a function parameter or let-binder \( e \in \text{Fin} \) then we eagerly force evaluation of \( e \) (the first rule), and otherwise we (lazily) substitute the relevant binder with the unevaluated parameter expression (the second, third, and fourth rule). The fifth rule evaluates the function in an application, as its value is always needed.

Constants

The final rule, application of a constant, requires the argument be reduced to a value; in a single step the expression is reduced to the output of the primitive constant operation.

Eager and Lazy Evaluation

To understand the role played by \( \text{Fin} \), consider an expression of the form \( (\lambda x.e) e_2 \). If \( e_2 \in \text{Fin} \) holds, then we eagerly force evaluation of \( e_2 \) (via the first rule), otherwise we \( \beta \)-reduce by substituting \( e_2 \) inside the body \( e \) (via the second rule). Note that if \( e_2 \) is already a value, then trivially, only the second rule can be applied. The same idea generalizes to the let and fix cases. Thus, we get lazy (call-by-name) and eager (call-by-value) semantics by instantiating \( \text{Fin} \) appropriately.

Serious and Trivial Expressions

Following Reynolds [26], we say that an expression \( e \) is trivial if \( e \not\rightarrow^* \text{crash} \) for some value \( v \neq \text{crash} \). Otherwise, we call the expression serious. That is, an expression is trivial iff it reduces to a non-crash value under CBN. Note the set of trivial expressions is closed under evaluation, i.e., if \( e \) is trivial and \( e \not\rightarrow e' \) (for any definition of \( \text{Fin} \)) then \( e' \) is also trivial.

Optimistic Evaluation

We define optimistic evaluation as the small-step relation \( \not\rightarrow_o \), obtained by defining

\[
e \in \text{Fin} \not\rightarrow e \text{ trivial } \land e \not\rightarrow \text{ not a value}
\]

Theorem 1 (Optimistic Equivalence).

\[
e \not\rightarrow^*_o c \iff e \not\rightarrow^*_n c.
\]

4. Type System

Next, we develop the static semantics for \( \lambda_4 \) and prove soundness with respect to the OPT, and hence CBN, evaluation strategies. We develop the type system in three steps. First, we present a general type system where termination is tracked abstractly via labels and show how to achieve sound refinement typing using a generic termination oracle (§4.1). Next, we describe a concrete instantiation of the oracle using refinements (§4.2). This decoupling allows us to make explicit the exact requirements for soundness, and also has the pragmatic benefit of leaving open the door for employing other kinds of termination analyses.

Basic Types

Figure 1 summarizes the syntax of \( \lambda_4 \) types. Basic types in \( \lambda_4 \) are natural numbers \( \text{nat} \) and booleans \( \text{bool} \). We include only two basic types for simplicity; it is straightforward to add any type whose values can be sized, and hence ordered using a well-founded relation, as discussed in §2.4.

Labels

We use two labels to distinguish terminating and diverging terms. Intuitively, the label \( \downarrow \) appears in types that describe expressions that always terminate, while the label \( \uparrow \) implies that the expressions may not terminate.

Types

Types in \( \lambda_4 \) include the standard basic refinement type \( b \) annotated with a label. The type \( \{ v: b \} e \) describes expressions of type \( b \) that satisfy the refinement \( c \). Intuitively, if the label is \( \downarrow \), then the expression definitely terminates, otherwise it may diverge. Finally, \( \lambda_4 \) types include dependent function types \( x:\tau \rightarrow \tau \).

Safety

We formalize safety properties by giving various primitive constants the appropriate refinement types. For example,

\[
3 : \{ v: \text{nat} \mid v = 3 \}
\]

(+) : \( x:\text{nat} \rightarrow y:\text{nat} \rightarrow \{ v:\text{nat} \mid v = x + y \} \)

(\_): \( x:\text{nat} \rightarrow \{ v:\text{nat} \mid v \neq 0 \} \rightarrow \text{nat} \)

error: \( \{ v:\text{nat} \mid \text{false} \} \rightarrow \tau \)

We assume that for any constant \( c \) with type

\[
T_Y(c) : x:\{ v: b \mid e \} \rightarrow \tau
\]

for any value \( v \), if the formula \( c \) is valid then the value \( [c](v) \) is not equal to \( \text{crash} \) and can be typed as \( \tau \). Thus, we define safety by requiring that a term does not evaluate to \( \text{crash} \).

Serious and Trivial Types

A type \( \tau \) is serious if \( \tau \in \{ v: b \mid e \} \), otherwise, it is trivial.

Notation

We write \( b \) to abbreviate the unrefined type \( \{ v: b \} \rightarrow \text{true} \). We ensure that serious types (which are, informally speaking, assigned to potentially diverging terms) are unrefined. We write \( b \) for \( b \), when label \( l \) can be either \( \downarrow \) or \( \uparrow \).
Well-Formedness

\[
\Gamma' = \text{Trivial}(\Gamma, v:b^i) \quad \quad \Gamma' \vdash e : \text{bool} \quad \text{WF-\text{↓}}
\]

\[
\Gamma \vdash \{ v:b^i \mid e \} \quad \text{WF-\text{↑}} \quad \Gamma \vdash \tau \quad \Gamma \vdash x:\tau \vdash \tau' \quad \text{WF-FUN}
\]

Subtyping

\[
\text{SmtValid}(\Gamma) \Rightarrow \{ e_1 \Rightarrow [e_2] \} \quad \Gamma \vdash \{ v:b^i \mid e_1 \} \preceq \{ v:b^i \mid e_2 \} \quad \leq \downarrow
\]

\[
\Gamma \vdash \{ v:b^i \mid e \} \preceq \{ v:b^i \mid \text{true} \} \quad \leq \uparrow
\]

\[
\Gamma \vdash \tau_2 \preceq \tau_1 \quad \Gamma, x:\tau_2 \vdash \tau_1 \preceq \tau_2' \quad \leq \text{-FUN}
\]

Typing

\[
\Gamma \vdash c : \text{Ty}(c) \quad \Gamma \vdash \{ x : \{ v:b^i \mid e \} \in \Gamma \} \quad \text{T-\text{CON}} \quad \Gamma \vdash \tau \quad \tau \not\in \{ v:b^i \mid e \} \quad \leq \downarrow
\]

\[
\Gamma, x:\tau \vdash e : \tau \quad \Gamma, x:\tau = e : \tau \quad \text{T-\text{VAR}}
\]

\[
\Gamma \vdash \lambda x.e : \tau \quad \Gamma \vdash \{ x : e \} \vdash \tau \quad \text{T-\text{FUN}}
\]

\[
\Gamma \vdash c_1 : \{ x : \tau \} \quad \Gamma \vdash c_2 : \tau \quad \text{T-\text{APP}}
\]

\[
\Gamma \vdash \text{let } x = e_1 \text{ in } e : \tau \quad \text{T-\text{LET}}
\]

\[
\Gamma, x:\tau =, f : x:\tau = \rightarrow \tau \vdash e : \tau \quad \Gamma \vdash x:\tau = \rightarrow \tau \quad \text{T-\text{REC}}
\]

Figure 3. Type-checking for \(\lambda_1\)

4.1 Type-checking

Next, we present the static semantics of \(\lambda_1\) by describing the type-checking judgments and rules. A type environment \(\Gamma\) is a sequence of type bindings \(x:\tau\). We use environments to define three kinds of rules: Well-formedness, Subtyping and Typing, which are mostly standard \[ \text{[19]} \]. The changes are that serious types are unrefined, and binders with serious types cannot appear in refinements.

Well-formedness A judgment \(\Gamma \vdash \tau\) states that the refinements in \(\tau\) are boolean values in the environment \(\Gamma\) restricted only to the binders whose types are trivial. The key rule is WF-\text{↓}, which checks that the refinement \(e\) of a trivial basic type is a boolean value under the environment \(\Gamma\) which contains all of the trivial bindings in \(\Gamma\) extended with the binding \(v:b^i\). To get the trivial bindings we use the function Trivial defined as:

\[
\text{Trivial}(\emptyset) \triangleq \emptyset \quad \text{Trivial}(x:\tau, \Gamma) \triangleq x:\tau, \text{Trivial}(\Gamma) \quad \text{if } \tau \text{ is trivial} \quad \text{Trivial}(x:\tau, \Gamma) \triangleq \text{Trivial}(\Gamma) \quad \text{otherwise}
\]

The rule WF-\text{↑} ensures serious types are unrefined.

Subtyping A judgment \(\Gamma \vdash \tau_1 \preceq \tau_2\) states that the type \(\tau_1\) is a subtype of the type \(\tau_2\) under environment \(\Gamma\). That is, informally speaking, when the free variables of \(\tau_1\) and \(\tau_2\) are bound to values described by \(\Gamma\), the set of values described by \(\tau_1\) is contained in the set of values described by \(\tau_2\). The interesting rules are the two rules that check subtyping of basic types. Rule \(\preceq \text{-}\uparrow\) checks subtyping on serious basic types, and requires that the supertype is unrefined. Rule \(\preceq \text{-}\downarrow\) checks subtyping on trivial basic types. As usual, subtyping reduces to implication checking: the embedding of the environment \(\Gamma\) strengthened with the interpretation of \(e_1\) in the logic should imply the interpretation of \(e_2\) in the refinement logic. The crucial difference is in the definition of embedding. Here, we keep only the trivial binders:

\[
[\Gamma] \triangleq \bigwedge \{ [e \mid v] \mid x : \{ v:b^i \mid e \} \in \text{Trivial}(\Gamma) \}
\]

Typing A judgment \(\Gamma \vdash e : \tau\) states that the expression \(e\) has the type \(\tau\) under environment \(\Gamma\). That is, when the free variables in \(e\) are bound to values described by \(\Gamma\), the expression \(e\) will evaluate to a value described by \(\tau\). Most of the rules are standard, we discuss only the interesting ones.

A variable expression \(x\) has a type if a binding \(x:\tau\) exists in the environment. The rule that is used for typing depends on the structure of \(\tau\). If \(\tau\) is basic and trivial, the rule \(T\text{-}\downarrow\) is used which, as usual, refines the basic type with the singleton refinement \(v = x\). Otherwise, the rule \(T\text{-VAR}\) is used to type \(x\) with \(\tau\). If \(\tau\) is basic and serious it should be unrefined, so its type is not strengthened with the singleton refinement.

The dependent application rule \(T\text{-APP}\) is standard: the type of the expression \(e_1 e_2\) is \(\tau\) where \(x\) is replaced with the expression \(e_2\). Note that if \(\tau_2\) is serious then \(e_2\) may diverge and so should not appear in any type. In this case, well-formedness ensures that \(\tau\) will not appear inside \(\tau\), and so \(\tau\) does not contain \(e_2\). Our system allows trivial arguments to be passed to a function that expects a serious input, but not the other way around.

Soundness With a Termination Oracle We prove soundness via preservation and progress theorems with respect to OPT evaluation, assuming the existence of a termination oracle that assigns labels to types so that serious expressions get serious types.

Hypothesis (Termination Oracle). If \(\emptyset \vdash e : \tau\) and \(\tau\) is a trivial type, then \(e\) is a trivial expression.

The proof can be found in [31], and relies on two facts: (1) We define constants so that the application of a constant is defined for every value (as required for progress) but preserves typing for values that satisfy their input refinements (as required for preservation). (2) Variables with serious types do not appear in any refinements. In particular, they do not appear in any type or environment; which makes it safe to trivially substitute them with any expression whilst preserving typing. With this, we can prove that application of a serious expressions preserves typing.

We combine preservation and progress to obtain the following result. The crash-freedom guarantee states that if a term \(c\) is well-typed, it will not crash under call-by-name semantics. From preservation and progress (and the fact that the constant crash has no type), we can show that a well-typed term will not crash under optimistic evaluation \(\langle \leftarrow o \rangle\). Hence, we can use Theorem 1 to conclude that the well-typed term cannot crash under lazy evaluation \(\langle \leftarrow n \rangle\). Similarly, type-preservation is translated from \(\leftarrow o\) to \(\leftarrow n\) by observing that a term reduces to a constant under \(\leftarrow n\) if it reduces to the same constant under \(\leftarrow o\).

Theorem 2 (Safety). Assuming the Termination Hypothesis,

- Crash-Freedom: If \(\emptyset \vdash e : c\) then \(\vdash e \leftarrow n\text{-}\text{crash}\).

- Type-Preservation: If \(\emptyset \vdash e : \tau\) and \(e \leftarrow n\text{-}\text{crash}\) then \(\emptyset \vdash e : \tau\).
4.2 Termination Analysis

Finally, we present an instantiation of the termination oracle that essentially generalizes the notion of sized types to the refinement setting, and hence complete the description of a sound refinement type system for λi. As described in §3, the key idea is to change the rule for typing recursive functions, so that the body of the function is checked under a termination-weakened environment where the function can only be called on smaller inputs.

Termination-Weakened Environments We formalize this idea by splitting the rule T-Rec that types fixpoints into the two rules shown in Figure 4. The rule T-Rec↑ can only be applied when the output type τ is serious, meaning that a recursive function typed with this rule can (when invoked) diverge. In contrast, rule T-Rec↓ is used to type a recursive function that always terminates, i.e., whose output type is trivial. The rule requires the argument x of such a function to be a trivial (terminating) natural number. Furthermore, when typing the body e, the type of the recursive function f : y.τ0 → τ is weakened to enforce that the function is invoked with an argument y that is strictly smaller than x at each recursive call-site. For clarity of exposition, we require that the decreasing metric be the value of the first parameter. It is straightforward to generalize the requirement to any well-founded metric on the arguments.

We prove that our modified type system satisfies the termination oracle hypothesis. That is, in λi, trivial types are only ascribed to trivial expressions. This discharges the Termination Hypothesis needed by Theorem 2 and proves safety of λi.

Theorem 3 (Termination). If ∅ ⊢ e : τ and τ is trivial then e is trivial.

The full proof is in [31], here we summarize the key parts.

Well-formed Terms We call a term e well-formed with respect to a type τ, written τ ⊢ e, if: (1) if τ = {v : b | e′} then e ↓v e′, for some value v, and (2) if τ = x:τ′ → τ then e ↓x e′, for some value v, and for all e′, such that ∅ ⊢ e′ : τ′ and τ′ |≡ e′, we have τ |≡ e′/x = e.

Well-formed Substitutions A substitution θ is either empty (∅) or of the form θ : [e/x]. A substitution θ is well-formed with respect to an environment Γ, written Γ ⊢ θ, if either both are empty, or the environment and the substitution are respectively Γ : x : τ and θ : [e/x], and (1) Γ ⊢ θ, (2) ∅ ⊢ θ : e, and (3) θ τ |≡ τ e. Now, we can connect types and termination.

Lemma 1 (Termination). If Γ ⊢ e : τ and Γ ⊢ τ |≡ τ e, then e ⊢ τ |≡ τ e.

The Termination Theorem 3 is an immediate corollary of Lemma 1 where Γ is the empty environment. Since we are using refinements to prove termination, our termination proof requires soundness and vice versa. We resolve this circularity by proving Preservation, Progress, and the Termination Lemma by mutual induction, to obtain Theorem 2 without the Termination Hypothesis [31].
not), is 4.5% of LOC. The specifications themselves are machine checkable versions of the comments placed around functions describing safe usage and behavior. Our default metric, namely the first parameter with an associated size measure, suffices to prove 67% of (recursive) functions terminating. 30% require a hint (i.e. the position of the decreasing argument) or a witness (3% required). Of the 18 functions marked as potentially diverging, we suspect 6 of the verification of T
 during the course of our evaluation, we present a brief overview of the low-level functions and interaction with unicode.

### 5.2 Case Study: Text

Next, to give a qualitative sense of the kinds of properties analyzed during the course of our evaluation, we present a brief overview of the verification of `Text`, which is the standard library used for serious unicode text processing in Haskell.

`Text` uses byte arrays and stream fusion to guarantee performance while providing a high-level API. In our evaluation of L\textsc{iquidHaskell} on `Text`\textsuperscript{[23]}, we focused on two types of properties: (1) the safety of array index and write operations, and (2) the functional correctness of the top-level API. These are both made more interesting by the fact that `Text` internally encodes characters using UTF-16, in which characters are stored in either two or four bytes. `Text` is a vast library spanning 39 modules and 5700 lines of code, however we focus on the 17 modules that are relevant to the above properties. While we have verified exact functional correctness size properties for the top-level API, we focus here on the low-level functions and interaction with unicode.

### Arrays and Texts

A `Text` consists of an (immutable) `Array` of 16-bit words, an offset into the `Text`, and a length describing the number of `Word16`s in the `Text`. The `Array` is created and filled using a `mutable` `MArray`. All write operations in `Text` are performed on `MArrays` in the ST monad, but they are `unsafe` before being used by the `Text` constructor. We write a measure denoting the size of an `MArray` and use it to type the write and freeze operations.

```haskell
measure malen :: MArray s -> Int
predicate EqLen A MA = alen A = malen MA
predicate OK I A = 0 <= I < malen A

\text{type } VO A = \{\text{Int} \mid \text{Ok } v A\}
```

```haskell
unsafeWrite :: m:MArray s -> V0 m -> Word16 -> ST s ()
unsafeFreeze :: m:MArray s -> ST s (V0 m)
```
Reasoning about Unicode. The function `writeChar` (abbreviating `UnsafeChar.unsafeWrite`) writes a Char into an MArray. TEXT uses UTF-16 to represent characters internally, meaning that every Char will be encoded using two or four bytes (one or two Word16s).

```
writeChar marr i c |
  n < 0x10000 = do
    unsafeWrite marr i (fromIntegral n) return 1
  otherwise = do
    unsafeWrite marr i lo unsafeWrite marr (i+1) hi
    return 2
where
  n = ord c
  m = n - 0x10000
  lo = fromIntegral $ (m `shiftR` 10) + 0xD800
  hi = fromIntegral $ (m .&. 0x3FF) + 0xDC00
```

The UTF-16 encoding complicates the specification of the function as we cannot simply require `i` to be less than the length of `marr` if `i` were `malen marr - 1` and `c` required two Word16s, we would perform an out-of-bounds write. We account for this subtlety with a predicate that states there is enough Room to encode `c`.

```
predicate OkN I A N = Ok (I+N-1) A
predicate Room I A C = if ord C < 0x10000 then OkN I A 1 else OkN I A 2
```

```
type OkSiz I A = {v:Nat | OkN I A v}
type OkChr I A = {v:Char | Room I A v}
```

Room `i` `marr` `c` says “if `c` is encoded using one Word16, then `i` must be less than `malen marr`, otherwise `i` must be less than `malen marr - 1`. ” `OkSiz I A` is an alias for a valid number of Word16s remaining after the index `I` of array `A`. `OkChr` specifies the `Chars` for which there is room (to write) at index `I` in array `A`. The specification for `writeChar` states that given an array `marr`, an index `i`, and a valid `Char` for which there is room at index `i`, the output is a monadic action returning the number of Word16 occupied by the char.

```
writeChar :: marr:MArrary s |
  -> i:Nat
  -> OkChr i marr
  -> ST s (OkSiz i marr)
```

**Bug** Thus, clients of `writeChar` should only call it with suitable indices and characters. Using LIQUIDHASKELL we found an error in one client, `mapAccumL`, which combines a map and a fold over a Stream, and stores the result of the map in a Text. Consider the inner loop of `mapAccumL`.

```
outer arr top = loop
  where
    loop !z !s !i =
      case next0 s of
        Done -> return (arr, (z,i))
        Skip s' -> loop z s' i
        Yield x s' |
          j => top -> do
            let top' = (top + 1) `shiftL` 1
            arr' <- new top'
            copyM arr' 0 arr 0 top
            outer arr' top' z s i
```

The error can be fixed by lifting `f z x` into the `where` clause and defining the write index `j` by comparing `ord c` (not `ord x`). LIQUIDHASKELL (and the authors) readily accepted our fix.

5.3 Code Changes

Our case studies also highlighted some limitations of LIQUIDHASKELL that we will address in future work. In most cases, we could alter the code slightly to facilitate verification. We briefly summarize the important categories here: refer to [3] for details.

**Ghost parameters** are sometimes needed in order to materialize values that are not needed for the computation, but are necessary to prove the specification; proving termination may require a decreasing value that is a function of several parameters. In future work it will be interesting to explore the use of advanced techniques for synthesizing ranking witnesses [8] to eliminate such parameters.

**Lazy binders** sometimes get in the way of verification. A common pattern in Haskell code is to define all local variables in a single `where` clause and use them only in a subset of all branches. LIQUIDHASKELL flags a few such definitions as `unsafe`, not realizing that the values will only be demanded in a specific branch. Currently, we manually transform the code by pushing binders inwards to the usage site. This transformation could be easily automated.

**Assumes** which can be thought of as “hybrid” run-time checks, had to be placed in a couple of cases where the verifier loses information. One source is the introduction of assumptions about mathematical operators that are currently conservatively modeled in the refinement logic (e.g. that multiplication is commutative and associative). These may be removed by using more advanced non-linear arithmetic decision procedures.

6. Related Work

Next we situate our work with closely related lines of research.

**Dependent Types** are the basis of many verifiers, or more generally, proof assistants. In this setting arbitrary terms may appear inside types, so to prevent logical inconsistencies, and enable the checking of type equivalence, all terms must terminate. “Full” dependently typed systems like Coq [4], Agda [23], and Idris [6] typically use various structural checks to ensure that all terms terminate. One can fake “lightweight” dependent types in Haskell [11,10,23]. In this style, the invariants are expressed in a restricted [16] total index language and relationships (e.g. $x < y$ and $y < z$) are combined (e.g. $x < z$) by explicitly constructing a term denoting the consequent from terms denoting the antecedents. On the plus side this “constructive” approach ensures soundness. It is impossible to witness inconsistencies, as doing so triggers diverging computations. However, it is unclear how easy it is to use restricted indices with explicitly constructed relations to verify the complex properties needed for large libraries.
Refinement Types are a form of dependent types where invariants are encoded via a combination of types and SMT-decidable logical refinement predicates \[\mathfrak{L}\]. Refinement types offer a highly automated means of verification and have been applied to check a variety of program properties, including functional correctness of data structures \[\mathfrak{L}\], security protocols \[\mathfrak{L}\], and compilers \[\mathfrak{L}\]. The language of refinements is restricted to ensure consistency, however, program variables (binders), or their singleton representatives \[\mathfrak{L}\], are crucially allowed in the refinements. As discussed in §2 this is sound under call-by-value evaluation, but under Haskell’s semantics any innocent binder can be potentially diverging, causing unsoundness. Finally, our implementation is based on LIQUID-HASKELL \[\mathfrak{L}\], which was unsound as it assumed CBV evaluation.

Size-based Termination Analyses have been used to verify termination of recursive functions, either using the “size-change principle” \[\mathfrak{L}\], or via the type system \[\mathfrak{L}\] by annotating types with size indices and verifying that the arguments of recursive calls have smaller indices. In work closely related to ours, Xi \[\mathfrak{L}\] encoded sizes via refinement types to prove totality of programs. What differentiates the above work from ours is that we do not aim to prove that all expressions converge; on the contrary, under a lazy setting diverging expressions are welcome. We use size analysis to track diverging terms in order to exclude them from the logic.

Static Checkers like ESCJava \[\mathfrak{L}\] are a classical way of verifying correctness through assertions and pre- and post-conditions. One can view Refinement Types as a type-based generalization of this approach. Classical contract checkers check “partial” (as opposed to “total”) correctness (i.e. safety) for eager, typically first-order, languages and need not worry about termination. We have shown that in the lazy setting, even “partial” correctness requires proving “total” correctness! \[\mathfrak{L}\] describes a static contract checker for Haskell that uses symbolic execution. The (checker’s) termination requires that recursive procedures only be unrolled up to some fixed depth. While this approach removes inconsistencies, it yields weaker, “bounded” soundness guarantees. Zeno \[\mathfrak{L}\] is another automatic prover for Haskell which proves properties by unrolling recursive definitions, rewriting, and goal-splitting, using sophisticated proof-search techniques to ensure convergence. As it is based on rewriting, “Zeno might loop forever” when faced with non-terminating functions, but will not conclude erroneous facts. Finally, \[\mathfrak{L}\] describes a novel contract checking technique that encodes Haskell programs into first-order logic. Intriguingly, the paper shows how the encoding, which models the denotational semantics of the code, is simplified by lazy evaluation.

Unlike the previous contract checkers for Haskell, our type-based approach does not rely on heuristics for unrolling recursive procedures, and instead uses SMT and abstract interpretation \[\mathfrak{L}\] to infer signatures, which we conjecture makes LIQUID-HASKELL more predictable. Of course, this requires LIQUID-HASKELL to be provided logical qualifiers (predicate fragments) which form the basis of the analysis’ abstract domain. In our experience, however, this is not an onerous burden as most qualifiers can be harvested from API specifications, and the overall workflow is predictable enough to enable the verification of large, real-world code bases.

References

[1] T. Ball, R. Majumdar, T. Millstein, and S. K. Rajaman. Automatic predicate abstraction of C programs. In PLDI, 2001.

[2] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. Mathematical Structures in Computer Science, 2004.

[3] J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. ACM TOPLAS, 2011.

[4] Y. Bertot and P. Castéran. CoqArt: The Calculus of Inductive Constructions. Springer Verlag, 2004.

[5] K. Bhargavan, C. Fournet, M. Kohlweiss, A. Pironti, and P.-Y Strub. Implementing tls with verified cryptographic security. In IEEE 5 & P, 2013.

[6] Edwin Brady. Idris: general purpose programming with dependent types. In PLPV, 2013.

[7] M. T. Chakravarty, G. Keller, and S. L. Peyton-Jones. Associated type synonyms. In ICFP, 2005.

[8] B. Cook, A. Podelski, and A. Rybalchenko. Termination proofs for systems code. In PLDI, 2006.

[9] J. Dunfield. Refined typechecking with Stardust. In PLPV, 2007.

[10] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In Haskell Symposium, 2012.

[11] R. Ennals and S. Peyton-Jones. Optimistic evaluation: An adaptive evaluation strategy for non-strict programs. In ICFP, 2003.

[12] C. Flanagan, K.R.M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In PLDI, 2002.

[13] C. Fournet, N. Swamy, J. Chen, P.-É. Dagand, P.-Y. Strub, and B. Livshits. Fully abstract compilation to javascript. In POPL, 2013.

[14] M. Greenberg, B. C. Pierce, and S. Weirich. Contracts made manifest. ICFP, 2012.

[15] J. Hughes, L. Pareto, and A. Sabry. Proving the correctness of reactive systems using sized types. In POPL, 1996.

[16] L. Jia, J. Zhao, V. Sjöberg, and S. Weirich. Dependent types and program equivalence. In POPL, 2010.

[17] Neil D. Jones and Nina Bohr. Termination analysis of the untyped lambda-calculus. In RTA, 2004.

[18] M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In PLDI, 2009.

[19] K.W. Knowles and C. Flanagan. Hybrid type checking. ACM TOPLAS, 2010.

[20] A. Mycroft. The theory and practice of transforming call-by-need into call-by-value. In ESOP, 1980.

[21] U. Norell. Towards a practical programming language based on dependent type theory. PhD thesis, Chalmers, 2007.

[22] B. O’Sullivan and T. Harper. text-0.11.2.3: An efficient packed unicode text type. http://hackage.haskell.org/package/text-0.11.2.3.

[23] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In ICFP, 2004.

[24] S. L. Peyton-Jones, D. Vytniotis, S. Weirich, and G. Washburn. Simple unification-based type inference for GADTs. In ICFP, 2006.

[25] Gordon Plotkin. Call-by-name, call-by-value and the lambda calculus. Theoretical Computer Science, 1975.

[26] John C. Reynolds. Definitional interpreters for higher-order programming languages. In 25th ACM National Conference, 1972.

[27] P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In PLDI, 2008.

[28] J. Rushby, S. Owre, and N. Shankar. Subtypes for specifications: Predicate subtyping in pvs. IEEE TSE, 1998.

[29] D. Sereni and N.D. Jones. Termination analysis of higher-order functional programs. In APLAS, 2005.

[30] W. Sonnex, S. Drossopoulou, and S. Eisenbach. Zeno: An automated prover for properties of recursive data structures. In TACAS, 2012.

[31] From Safety to Termination and back: SMT-Based verification for Lazy languages. Supplementary Materials.

[32] A. M. Turing. On computable numbers, with an application to the entscheidungsproblem. In LMS, 1936.

[33] H. Unno, T. Terauchi, and N. Kobayashi. Automating relatively complete verification of higher-order functional programs. In POPL, 2013.

[34] N. Vazou, P. Rondon, and R. Jhala. Abstract refinement types. In ESOP, 2013.
[35] D. Vytiniotis, S.L. Peyton-Jones, K. Claessen, and D. Rosén. Halo: haskell to logic through denotational semantics. In POPL, 2013.
[36] P. Wadler. Call-by-value is dual to call-by-name. In ICFP, 2003.
[37] H. Xi. Dependent types for program termination verification. In LICS, 2001.
[38] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In PLDI, 1998.
[39] Dana N. Xu, Simon L. Peyton-Jones, and Koen Claessen. Static contract checking for haskell. In POPL, 2009.