Technical Report: Refinement Types For Haskell

Niki Vazou Eric L. Seidel Ranjit Jhala
UC San Diego

Dimitrios Vytiniotis Simon Peyton-Jones
Microsoft Research

Abstract

SMT-based refinement typing is unsound under lazy evaluation. When checking an expression, refinement systems implicitly assume that all the free variables in the expression are bound to *values*. This property is trivially guaranteed by eager, but does not hold under lazy, evaluation. Thus, to be sound and precise, a refinement type system for Haskell must reason about *which subset of* binders actually reduces to values. We present a stratified type system that labels binders as potentially diverging or not, and that (circularly) uses refinement types to verify the labeling. We have implemented our system in LIQUIDHASKELL and present an experimental evaluation of our approach on more than 10,000 lines of widely used Haskell libraries. We show that LIQUIDHASKELL is able to prove 96% of all recursive functions terminating, while requiring a modest 1.7 lines of termination-annotations per 100 lines of code.

1. Introduction

Refinement types are unsound under Haskell's lazy semantics. Refinement types encode invariants by composing types with SMT-decidable refinement predicates [31, 41], generalizing Floyd-Hoare Logic (*e.g.* EscJava [16]) for functional languages. For example

```
type Pos = {v:Int | v >  0}
type Nat = {v:Int | v >= 0}
```

are the basic type `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

```
div :: n:Nat -> d:Pos -> {v:Nat | v <= n}
```

which states that the function `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 `div` statically typechecks, we can rest assured that executing the program will not lead to any unpleasant divide-by-zero errors. By combining types and SMT based validity checking, refinement types have automated the verification of programs with recursive datatypes, higher-order functions, and polymorphism. Several groups have used refinements to statically verify properties ranging from simple array safety [30, 41] to functional correctness of data structures [22], security protocols [5], and compiler correctness [35].

Given the remarkable effectiveness of the technique, we embarked on the project of developing a refinement type based verifier for Haskell. The previous 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. Our first contribution is to show that standard refinement systems crucially rely on a property of eager languages: when analyzing any term, one can assume that *all* the free variables appearing in the term are bound to *values*. This property lets us check each term in an environment where the free variables are logically constrained according to their refinements. Unfortunately, this property does not hold for lazy evaluation, where free variables can be lazily substituted with arbitrary (potentially diverging) expressions, which breaks soundness (§ 2).

The two natural paths towards soundness are blocked by challenging problems. The first path is to *conservatively ignore* free variables except those that are guaranteed to be values *e.g.* by pattern matching, `seq` or strictness annotations. While sound, this leads to a drastic loss of precision. The second path is to *explicitly* reason about divergence within the refinement logic. While sound and precise, this route makes the refinement logic three-valued, making it impossible to use existing SMT machinery (§ 8).

Our second contribution, is a novel approach that enables sound and precise checking with existing SMT solvers, using a *stratified* type system that labels binders as potentially diverging or not (§ 4). While previous stratified systems [11] would suffice for soundness, we show how to recover precision by using refinement types to develop a notion of *terminating fixpoint* combinators that allows the type system to automatically verify that a wide variety of recursive functions actually terminate (§ 5).

Our third contribution is an extensive empirical evaluation of our approach on more than $10,000$ lines of widely used complex Haskell libraries. We have implemented our approach in LIQUIDHASKELL, an SMT based verifier for Haskell. LIQUIDHASKELL is able to prove 96% of all recursive functions terminating, requiring a modest 1.7 lines of termination annotations per 100 lines of code, thereby enabling the sound, precise, and automated verification of functional correctness properties of real-world Haskell codes (§ 6).

2. Overview

We start with an overview of our contributions. After recapitulating the basics of *refinement types* we illustrate why the classical approach based on *verification conditions* (VCs) is *unsound* due to lazy evaluation. Next, we step back to understand precisely how the VCs arise from refinement *subtyping*, and how subtyping is different under eager and lazy evaluation. In particular, we demonstrate that under lazy (but *not* eager) evaluation, the refinement type system, and hence the VCs, must explicitly account for divergence. Consequently, we develop a type system that accounts for divergence in a modular and syntactic fashion, and illustrate its use via several small examples. Finally, we show how a refinement-based termination analysis can be used to improve precision, yielding a highly effective SMT-based verifier for Haskell.

2.1 Standard Refinement Types: From Subtyping to VC

First, let us see how standard refinement type systems [23, 30] will use the refinement type aliases `Pos` and `Nat` and the specification for `div` from § 1 to *reject* `bad` and *accept* `good`.

```
bad      :: Nat -> Nat -> Int
bad x y  = x `div` y
```

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

Refinement Subtyping To analyze the body of `bad`, the refinement type system will check that the second parameter `y` has type `Pos` at the call to `div`; formally, that the actual parameter `y` is a *subtype* of the type of `div`'s second input, via a subtyping query:

$$\mathtt{x} : \{\mathtt{x} \geq 0\},\ \mathtt{y} : \{\mathtt{y} \geq 0\} \vdash \{\mathtt{v} \geq 0\} \preceq \{\mathtt{v} > 0\}$$

Verification Conditions To discharge the above subtyping query, a refinement type system, generalizing the classical Floyd-Hoare Logic [18], generates a *verification condition* (VC). A VC is a logical formula that stipulates that under the assumptions corresponding to the environment bindings, the refinement in the sub-type *implies* the refinement in the super-type.

$$(\mathtt{x} \geq 0) \wedge (\mathtt{y} \geq 0) \ \Rightarrow\ (\mathtt{v} \geq 0) \Rightarrow (\mathtt{v} > 0)$$

Refinement type systems are carefully engineered (§ 4) so that (*unlike* with full dependent types) the logic of refinements *precludes* arbitrary functions and only includes formulas from efficiently decidable logics, *e.g.* the quantifier-free logic of linear arithmetic and uninterpreted functions (QFLIA), and so VCs like the above can be efficiently validated by SMT solvers [13]. In this case, the solver will reject the above VC as *invalid* meaning the implication, and hence, the relevant subtyping requirement does not hold. So the refinement type system will *reject* `bad`.

On the other hand, a refinement system *accepts* `good`. Here, the subtyping query for the argument to `div` is

$$\mathtt{x} : \{\mathtt{x} \geq 0\},\ \mathtt{y} : \{\mathtt{y} \geq 0\} \vdash \{\mathtt{v} = \mathtt{y} + 1\} \preceq \{\mathtt{v} > 0\} \quad (1)$$

which reduces to the *valid* VC

$$(\mathtt{x} \geq 0) \wedge (\mathtt{y} \geq 0) \ \Rightarrow (\mathtt{v} = \mathtt{y} + 1) \Rightarrow (\mathtt{v} > 0) \quad (2)$$

2.2 Lazy Evaluation Makes VCs Unsound

Lazy evaluation renders the above technique unsound. Consider

```
diverge   :: Int -> {v:Int | false}
diverge n = diverge n
```

The output type captures the *post-condition* that the function returns an `Int` satisfying `false`. This counter-intuitive specification states, in essence, that the function *does not terminate*, *i.e.* does not return *any* value. Any standard refinement type checker (or Floyd-Hoare verifier like Dafny[1]) will verify the given signature for `diverge` via the classical method of inductively *assuming* the signature holds for `diverge` and then *guaranteeing* the signature [18, 25]. Next, consider the call to `div` in `explode`:

```
explode   :: Int -> Int
explode x = let {n = diverge 1; y = 0}
            in  x `div` y
```

To analyze `explode`, the refinement type system will check that `y` has type `Pos` at the call to `div`, *i.e.* will check that

$$\mathtt{n} : \{\mathtt{false}\},\ \mathtt{y} : \{\mathtt{y} = 0\} \vdash \{\mathtt{v} = 0\} \preceq \{\mathtt{v} > 0\} \quad (3)$$

In the subtyping environment `n` is bound to the type corresponding to the *output* type of `diverge`, and `y` is bound to the singleton type stating `y` equals `0`. In this environment, we must prove that actual parameter's type – *i.e.* that of `y` – is a subtype of `Pos`. The subtyping, as previously 2.1 discussed, reduces to the VC:

$$\mathtt{false} \wedge \mathtt{y} = 0 \ \Rightarrow (\mathtt{v} = 0) \Rightarrow (\mathtt{v} > 0) \quad (4)$$

The SMT solver proves this VC valid by using the contradiction in the antecedent, thereby unsoundly proving the call to `div` safe!

[1] http://rise4fun.com/Dafny/wVGc

Eager vs. Lazy Verification Conditions At this point, we pause to emphasize that the problem lies in the fact that the classical technique for encoding subtyping (or generally, Hoare's "rule of consequence" [18]) with VCs is *unsound under lazy evaluation*. To see this, observe that the VC (4) is perfectly *sound* under eager (strict, call-by-value) evaluation. In the eager setting, the program is safe in that `div` is never called with the divisor `0`, as it is not called at all! The inconsistent antecedent in the VC logically encodes the fact that, under eager evaluation, the call to `div` is *dead code*. Of course, this conclusion is spurious under Haskell's lazy semantics! As `n` is not required, the program will dive headlong into evaluating the `div` and hence crash, rendering the VC meaningless.

The Problem is Laziness (Not Recursion) Readers familar with fully dependently typed languages like Cayenne [2], Agda [26], Coq [6], or Idris [8], may be tempted to attribute the unsoundness to the presence of arbitrary recursion and hence non-termination (*e.g.* in `diverge`). In dependently typed languages, arbitrary terms may appear in types. Thus, arbitrary recursion can lead to non-terminating functions that make it impossible to define the semantics of types, and to check type equivalence.

However, in the refinement setting, recursion is *not* the problem. The types are carefully engineered to *not* contain arbitrary terms, but only contain formulas from restricted logics that preclude arbitrary user-defined functions [15, 35, 41]. Thus, termination has never been an issue with refinement type systems. Indeed, we will show how to make refinement types sound under laziness *without* restricting recursion or requiring proofs of termination.

2.3 Semantics, Subtyping & Verification Conditions

To understand the problem, let us take a step back to get a clear view of the relationship between the operational semantics, subtyping, and verification conditions. We use the formulation of evaluation-order independent refinement subtyping developed for λ^H [23] in which refinements are *arbitrary* expressions. We define a denotation for types and use it to define subtyping declaratively.

Denotations of Types and Environments Recall `Pos` defined as $\{\mathtt{v}{:}\mathtt{Int} \mid 0 < \mathtt{v}\}$. Intuitively, `Pos` denotes the *set of* `Int` expressions which evaluate to values greater than `0`. We formalize this intuition by defining the denotation of a type as:

$$[\![\{x{:}t \mid p\}]\!] \doteq \{e \mid \emptyset \vdash e : t,\ \text{if } e \hookrightarrow^* v \text{ then } p\,[v/x] \hookrightarrow^* \mathtt{true}\}$$

That is, the type denotes the set of expressions e that have the corresponding base type t which *diverge or* reduce to values that make the refinement reduce to `true`. Thus, quoting [23], "refinement types specify partial and not total correctness".

An *environment* Γ is a sequence of type bindings, and a *closing substitution* θ is a sequence of expression bindings:

$$\Gamma \doteq x_1{:}\tau_1, \ldots x_n{:}\tau_n \qquad \theta \doteq x_1 \mapsto e_1, \ldots, x_n \mapsto e_n$$

Thus, we define the denotation of Γ as the set of substitutions:

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

Declarative Implication & Subtyping Equipped with interpretations for types and environments, we define the *declarative implication* T-IMP rule between refinements:

$$\frac{\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \theta(p_1) \hookrightarrow^* \mathtt{true} \Rightarrow \theta(p_2) \hookrightarrow^* \mathtt{true}}{\Gamma \vdash p_1 \Rightarrow p_2}\ \text{T-IMP}$$

Finally, we define declarative subtyping D-SUB (over base types B) to be declarative implication between the refinements:

$$\frac{\Gamma, x{:}B \vdash p_1 \Rightarrow p_2}{\Gamma \vdash \{x{:}B \mid p_1\} \preceq \{x{:}B \mid p_2\}}\ \text{D-SUB}$$

Declarative Subtyping with Lazy Evaluation Let us revisit the query (3) to see whether it holds under the declarative subtyping

rule D-SUB. The query reduces to the declarative implication:

$$\texttt{n} : \{\texttt{false}\},\ \texttt{y} : \{\texttt{y} = 0\},\ \texttt{v} : \texttt{Int} \vdash v = 0 \Rightarrow v > 0 \quad (5)$$

This implication *does not* hold, as shown by θ that maps n to *any diverging* expression of type `Int`, and y and v to the value 0.

Declarative Subtyping with Eager Evaluation Since the implication (5) is *invalid*, λ^H cannot verify `explode` under eager evaluation. However, Belo *et al.* [4] note that under eager (call-by-value) evaluation, each binder in the environment is only added *after* the previous binders have been reduced to *values*. Hence, under eager evaluation we can *restrict the range* of the closing substitutions to values (as opposed to expressions). Let us reconsider (5) in this new light: there *is no value* that we can map n to, so the set of denotations of the environment is empty. Hence, the implication (5) vacuously holds under eager evaluation, which proves the program safe. Belo's observation is implicitly used by refinement types for eager languages to prove that the standard (as presented in 2.1) reduction from subtyping to VC is sound.

Algorithmic Subtyping via Verification Conditions The above subtyping (D-SUB) and implication (T-IMP) rules allow us to prove preservation and progress [23] but quantify over evaluation of arbitrary expressions, and so are statically undecidable. To make checking *algorithmic* we approximate the implication using *verification conditions* (VCs), formulas drawn from a decidable logic, that are valid only if the undecidable implication holds. As we have seen, the classical VC is sound only under eager evaluation. Next, let us use the distinctions between lazy- and eager- declarative implication, to obtain sound, decidable VCs for the lazy setting.

Step 1: Restricting Refinements To Decidable Logics Given that in λ^H refinements can be *arbitrary* expressions, the first step towards obtaining a VC, regardless of evaluation order, is to restrict the refinements to a *decidable* logic. We choose the quantifier free logic of equality, uninterpreted functions, linear arithmetic, and data constructors (QF-EUFLIAD). We carefully design our typing rules to ensure that for any derivation tree, starting with types refined from the language of the decidable logic, the refinements of all intermediate types are also restricted to this language.

Step 2: Translating Implications into VCs Recall that our goal is to encode the antecedent of T-IMP

$$\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \theta(p_1) \hookrightarrow^* \texttt{true} \Rightarrow \theta(p_2) \hookrightarrow^* \texttt{true}$$

as a logical formula, that is valid *only when* the above holds. Intuitively, we can think of the closing substitutions θ as corresponding to *assignments* or *interpretations* $(\!|\theta|\!)$ of variables X of the VC. Thus, we will translate the above antecedent into a VC

$$\forall X.(\!|\Gamma|\!) \Rightarrow (\!|p_1|\!) \Rightarrow (\!|p_2|\!)$$

where $(\!|\Gamma|\!)$ and $(\!|p_i|\!)$ are the translation of the environment, and refinements into logical formulas that are only satisfied by assignments $(\!|\theta|\!)$ that respectively correspond to closing substitutions $\theta \in [\![\Gamma]\!]$ and $\theta(p_i) \hookrightarrow^* \texttt{true}$. As refinements p_i are from the logic, they can be trivially translated into formulas, *i.e.* $(\!|p_i|\!) \doteq p_i$. We translate environments by conjoining their bindings:

$$(\!|x_1{:}\tau_1, \ldots, x_n{:}\tau_n|\!) \doteq (\!|x_1{:}\tau_1|\!) \wedge \ldots \wedge (\!|x_n{:}\tau_n|\!)$$

However, since types denote *partial correctness*, the translations must also explicitly account for possible divergence:

$$(\!|x{:}\{v{:}B \mid p\}|\!) \doteq \text{“}x \text{ is a value”} \Rightarrow p\,[x/v]$$

That is, we *cannot* assume that each x satisfies its refinement p; we must *guard* that assumption with a predicate stating that x is bound to a value (not a diverging term.)

A crucial question that arises is *how* and *when* can one discharge these guards to conclude that x indeed satisfies p. One natural route is to enrich the refinement logic with a predicate that states that “x is a value”, and then use the SMT solver to *explicitly* reason about this predicate and hence, divergence. Unfortunately, we show in § 8, that such predicates lead to three-valued logics, and hence, are *outside* the scope of the efficiently decidable theories supported by current solvers. Hence, this route is problematic if we want to use *existing* SMT machinery to build automated verifiers for Haskell.

2.4 Our Answer: Implicit Reasoning About Divergence

One way forward is to *implicitly* reason about divergence by *eliminating* the “x is a value” guards (*i.e. value guards*) from the VCs.

Implicit Reasoning: Eager Evaluation Under eager evaluation the domain of the closing substitutions can be restricted to values [4]. Thus, we can trivially eliminate the value guards, as they are guaranteed to hold by virtue of the evaluation order! Returning to `explode`, we see that after eliminating the value guards, we get the VC (4) which is, therefore, sound under eager evaluation.

Implicit Reasoning: Lazy Evaluation However, with lazy evaluation, we cannot just eliminate the value guards, as the closing substitutions are not restricted to just values. Our solution is to take this reasoning out of the hands of the SMT logic and place it in the hands of a *stratified type system* in which each type is labeled as: A Div-type, written τ, which are the default types given to binders that *may diverge*, or, a Wnf-type, written $\tau^{\downarrow}$, which are given to binders that are guaranteed to reduce to *Haskell values*, *i.e.* to Weak Head Normal Form (WHNF), or, a Fin-type, written $\tau^{\Downarrow}$, which are given to binders that are guaranteed to reduce to *finite values*. This stratification lets us generate VCs that are sound for lazy evaluation. The key piece is the translation of environment bindings:

$$(\!|x{:}\{v{:}B \mid p\}|\!) \doteq \begin{cases} \texttt{true}, & \text{if } B \text{ is a Div type} \\ p\,[x/v]\,, & \text{otherwise} \end{cases}$$

That is, if the binder may diverge, we simply *omit* any constraints for it in the VC, and otherwise the translation directly states (*i.e.* without the value guard) that the refinement holds. Returning to `explode`, the subtyping query (3) yields the *invalid* VC

$$\texttt{true} \Rightarrow v = 0 \Rightarrow v > 0$$

and so `explode` is soundly rejected under lazy evaluation.

2.5 Verification With Stratified Types

While it is reassuring that the lazy VC soundly *rejects* unsafe programs like `explode`, we now demonstrate by example that it usefully *accepts* safe programs. First, we show how the basic system – all terms have Div types – allows us to prove “partial correctness” properties without requiring termination. Second, we show how to extend the basic system by using Haskell's pattern matching semantics to assign the pattern match scrutinees Wnf types, thereby increasing the expressiveness of the verifier. Third, we show how to further improve the precision and usability of the system by using a termination checker to assign various terms Fin types. Fourth, we close the loop, by illustrating how the termination checker can itself be realized using refinement types.

Example 1: VCs and Partial Correctness The first example illustrates how, unlike Curry-Howard based systems, refinement types *do not require* termination. That is, we retain the Floyd-Hoare notion of “partial correctness”, and can verify programs where *all* terms have Div-types. Consider `ex1` which uses the result of `collatz` as a divisor.

```
ex1   :: Int -> Int
ex1 n = let x = collatz n in 10 `div` x

collatz :: Int -> {v:Int | v = 1}
```

```
collatz n
  | n == 1    = 1
  | even n    = collatz (n / 2)
  | otherwise = collatz (3*n + 1)
```

The jury is still out on *whether* the `collatz` function terminates [1], but it is easy to verify that its output is a Div `Int` equal to `1`. At the call to `div` the parameter `x` has the output type of `collatz`, yielding the subtyping query:

$$\texttt{x}{:}\{v{:}\texttt{Int} \mid v = 1\} \vdash \{\texttt{v} = \texttt{1}\} \preceq \{\texttt{v} > \texttt{0}\}$$

where the sub-type is just the type of `x`. As `Int` is a Div type, the above reduces to the VC $(\texttt{true} \Rightarrow v = 1 \Rightarrow v > 0)$ which the SMT solver proves valid, thereby verifying `ex1`.

Example 2: Improving Precision By Pattern Matching If all binders in the environment have Div-types then, effectively, the verifier can make *no* assumptions about the context in which a term evaluates, which leads to a drastic loss of precision. Consider:

```
head    :: {v:[a] | not (emp v)} -> a
head xs = case xs of
            (x:_) -> x
            []    -> error "yikes"

error   :: {v:String | false} -> a
error   = undefined
```

where `emp` is a *measure* [22] (or logical function [35]):

```
measure emp :: [a] -> Prop
emp []      = true
emp (x:xs)  = false
```

`head` is safe as its input type stipulates that it will only be called with lists that are *not* `[]`, and so `error "..."` is dead code. However, the call to `error` generates the subtyping query

$$\begin{array}{l}\texttt{xs}{:}\{\texttt{xs}{:}[a] \mid \neg(\texttt{emp xs})\} \\ \texttt{b}{:}\{\texttt{b}{:}[a] \mid (\texttt{emp xs})\}\end{array} \vdash \{\texttt{true}\} \preceq \{\texttt{false}\}$$

The match-binder `b` holds the result of the match [34]. In the `[]` case, we assign it a refinement $(\texttt{emp xs})$ as the measure is `true` for empty lists [22]. The verifier would *reject* the program as the above subtyping reduces to the invalid VC $(\texttt{true} \Rightarrow \texttt{true} \Rightarrow \texttt{false})$.

We address this problem by observing that a pattern match forces evaluation. Hence, inside each case of the **`case`**-**`of`** expression, the scrutinee and match binder are guaranteed to be Haskell values in WHNF. This intuition is formalized by the typing rule (T-CASE), which checks each case after assuming the scrutinee and the match binder have Wnf types. With this optimization, the call to `error` yields the subtyping query:

$$\begin{array}{l}\texttt{xs}{:}\{\texttt{xs}{:}[a]^{\downarrow} \mid \neg(\texttt{emp xs})\} \\ \texttt{b}{:}\{\texttt{b}{:}[a]^{\downarrow} \mid (\texttt{emp xs})\}\end{array} \vdash \{\texttt{true}\} \preceq \{\texttt{false}\}$$

That is, both `xs` and `b` have Wnf types. Now, the verifier *accepts* the program as the above subtyping reduces to the valid VC

$$\neg(\texttt{emp xs}) \wedge (\texttt{emp xs}) \Rightarrow \texttt{true} \Rightarrow \texttt{false}$$

The above method also applies to terms that have been `seq`-ed or have strictness annotations. Consequently, our system can naturally support idiomatic Haskell, *e.g.* taking the `head` of an infinite list:

```
ex2 x    = head (repeat x)

repeat   :: a -> {v:[a] | not (emp v)}
repeat y = y : repeat y
```

Example 3: Improving Precision By Termination While pattern matching allows us to ensure that certain environment binders have non-Div types, the system still lacks expressivenes leading to many false alarms (§ 6). For example, consider:

```
ex3 = let {x = 1; y = inc x} in 10 `div` y

inc :: z:Int -> {v:Int | v > z }
inc = \z -> z + 1
```

The call to `div` in `ex3` is obviously safe, but the system would reject it, as the call yields the subtyping query:

$$\texttt{x}{:}\{\texttt{x}{:}\texttt{Int} \mid \texttt{x} = \texttt{1}\},\ \texttt{y}{:}\{\texttt{y}{:}\texttt{Int} \mid \texttt{y} > \texttt{x}\} \vdash \{\texttt{v} > \texttt{x}\} \preceq \{\texttt{v} > \texttt{0}\}$$

Which, as `x` is a Div type, reduces to the invalid VC

$$\texttt{true} \Rightarrow v > \texttt{x} \Rightarrow v > 0$$

We could solve the problem by matching against `x` but that would lead to rather ugly, and non-idiomatic code.

Instead, our next key optimization is based on the observation that in practice, *most terms don't diverge*. Thus, we can use a termination analysis to aggressively assign terminating expressions Fin types, which lets us strengthen the environment assumptions needed to prove the VCs. For example, the term `1` obviously terminates. Hence, we type `x` as $\texttt{Int}^{\Downarrow}$, yielding the subtyping query:

$$\begin{array}{l}\texttt{x} : \{\texttt{x}{:}\texttt{Int}^{\Downarrow} \mid \texttt{x} = \texttt{1}\} \\ \texttt{y} : \{\texttt{y}{:}\texttt{Int} \mid \texttt{y} > \texttt{x}\}\end{array} \vdash \{\texttt{v} > \texttt{x}\} \preceq \{\texttt{v} > \texttt{0}\} \tag{6}$$

As `x` is Fin, we accept `ex3` by proving the validity of the VC

$$\texttt{x} = 1 \Rightarrow v > \texttt{x} \Rightarrow v > 0 \tag{7}$$

Example 4: Verifying Termination With Refinements While it is straightforward to conclude that the term `1` does not diverge, how do we do so in general? For example:

```
ex4 = let {x = f 9; y = inc x} in 10 `div` y

f   :: Nat -> {v:Int | v = 1}
f n = if n == 0 then 1 else f (n-1)
```

We check the call to `div` via subtyping query (6) and VC (7), which requires us to prove that `f` terminates on *all* $\texttt{Nat}^{\Downarrow}$ inputs.

We solve this problem by showing how refinement types may themselves be used to prove termination, by following the classical recipe of proving termination via decreasing metrics [36] as embodied in sized types [19, 40]. The key idea is to show that each recursive call is made with arguments of a *strictly smaller* size, where the size is itself a well founded metric, *e.g.* a natural number.

We formalize this intuition by type checking recursive procedures in a termination-weakened environment where the procedure itself may only be called with arguments that are strictly smaller than the current parameter (T-REC-↑). For example, to prove `f` terminates, we check its body in an environment

$$\texttt{n} : \texttt{Nat}^{\Downarrow} \qquad \texttt{f} : \{\texttt{n}'{:}\texttt{Nat}^{\Downarrow} \mid \texttt{n}' < \texttt{n}\} \rightarrow \{\texttt{v} = \texttt{1}\}$$

where we have weakened the type of `f` to stipulate that it *only* be (recursively) called with `Nat` values $\texttt{n}'$ that are *strictly less than* the (current) parameter `n`. The body type-checks as the recursive call generates the valid VC

$$0 \leq \texttt{n} \wedge \neg(0 = \texttt{n}) \Rightarrow v = \texttt{n} - 1 \Rightarrow (0 \leq v < \texttt{n})$$

3. Declarative Typing: λ^U

Next, we formalize our stratified refinement type system, in two steps. First, in this section, we present a core calculus λ^U, with a general β-reduction semantics. We describe the syntax, operational semantics, and sound but undecidable declarative typing rules for λ^U. Second, in § 4, we describe QF-EUFLIAD, a subset of λ^U that forms a decidable logic of refinements, and use it to obtain λ^D with decidable SMT-based algorithmic typing.

$$
\begin{array}{rrcl}
\textbf{\textit{Constants}} & c & ::= & 0, 1, -1, \ldots \mid \texttt{true}, \texttt{false} \\
 & & \mid & +, -, \ldots \mid =, <, \ldots \mid \texttt{crash} \\
\textbf{\textit{Values}} & v & ::= & c \mid \lambda x.e \mid D\ \overline{e} \\
\textbf{\textit{Expressions}} & e & ::= & v \mid x \mid e\ e \mid \texttt{let}\ x = e\ \texttt{in}\ e \\
 & & \mid & \texttt{case}\ x = e\ \texttt{of}\ \{D\ \overline{x} \rightarrow e\} \\
\textbf{\textit{Basic Types}} & B & ::= & \texttt{int} \mid \texttt{bool} \mid T \\
\textbf{\textit{Types}} & \tau & ::= & \{v{:}B \mid e\} \mid x{:}\tau \rightarrow \tau \\
\hline
\textbf{\textit{Contexts}} & C & ::= & \bullet \mid C\ e \mid c\ C \mid D\ \overline{e}\ C\ \overline{e} \\
 & & \mid & \texttt{case}\ x = C\ \texttt{of}\ \{D\ \overline{y} \rightarrow e\}
\end{array}
$$

Reduction $\boxed{e \hookrightarrow e}$

$$
\begin{array}{rcll}
C[e] & \hookrightarrow & C[e'] & \text{if } e \hookrightarrow e' \\
c\ v & \hookrightarrow & \delta(c, v) & \\
(\lambda x.e)\ e_x & \hookrightarrow & e\,[e_x/x] & \\
\texttt{let}\ x = e_x\ \texttt{in}\ e & \hookrightarrow & e\,[e_x/x] & \\
\texttt{case}\ x = D_j\ \overline{e}\ \texttt{of}\ \{D_i\ \overline{y_i} \rightarrow e_i\} & \hookrightarrow & e_j\,[D_j\ \overline{e}/x]\,[\overline{e}/\overline{y_j}] &
\end{array}
$$

Figure 1. λ^U: Syntax and Operational Semantics

3.1 Syntax

Figure 5 summarizes the syntax of λ^U, which is essentially the calculus λ^H [23] *without* the dynamic checking features (like casts), but *with* the addition of data constructors.

Constants The primitive constants of λ^U include `true`, `false`, `0`, `1`, -1, *etc.*, and arithmetic and logical operators like $+$, $-$, $\leq$,/, $\wedge$, $\neg$. In addition, we include a special *untypable* constant `crash` that models "going wrong". Primitive operations return a `crash` when invoked with inputs outside their domain, *e.g.* when / is invoked with `0` as the divisor, or when `assert` is applied to `false`.

Data Constructors We encode data constructors as special constants. Each data type has an arity $\mathsf{Arity}(T)$ that represents the exact number of data constructors that return a value of type T. For example the data type `NList`, which represents lists of integers, has two data constructors: `NNull` and `NCons`, *i.e.* has arity 2.

Values & Expressions The values of λ^U include constants, λ-abstractions $\lambda x.e$, and fully applied data constructors D that wrap expressions. The expressions of λ^U include values, as well as variables x, applications $e\ e$, and the `case` and `let` expressions.

3.2 Operational Semantics

Figure 5 summarizes the small step contextual β-reduction semantics for λ^U. Note that we allow for reductions under data constructors, and thus, values may be further reduced. We write $e \hookrightarrow^j e'$ if there exist $e_1, \ldots, e_j$ such that e is e_1, e' is e_j and $\forall i, j, 1 \leq i < j$, we have $e_i \hookrightarrow e_{i+1}$. We write $e \hookrightarrow^* e'$ if there exists some (finite) j such that $e \hookrightarrow^j e'$.

Constants 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. For example, consider $=$, the primitive equality operator on integers. We have $\delta(=, n) \doteq =_n$ where $\delta(=_n, m)$ equals `true` iff m is the same as n.

3.3 Types

λ^U types include basic types, which are *refined* with predicates, and dependent function types. *Basic types* B comprise integers, booleans, and a family of data-types T (representing lists, trees *etc.*) For example the data type `NList` represents lists of integers. We refine basic types with predicates (boolean valued expressions e) to obtain *basic refinement types* $\{v{:}B \mid e\}$. Finally, we have dependent *function types* $x{:}\tau_x \rightarrow \tau$ where the input x has the type τ_x and the output τ may refer to the input binder x.

Notation We write B to abbreviate $\{v{:}B \mid \texttt{true}\}$, and $\tau_x \rightarrow \tau$ to abbreviate $x{:}\tau_x \rightarrow \tau$ if x does not appear in τ_x or τ. We use p, q, and r for refinements, and use _ for unused binders. We write $\{v{:}\texttt{nat}^l \mid p\}$ to abbreviate $\{v{:}\texttt{int}^l \mid 0 \leq v \wedge p\}$.

Denotations Each type τ *denotes* a set of expressions $[\![\tau]\!]$, that are defined via the dynamic semantics [23]. Let $\lfloor\tau\rfloor$ be the type we get if we erase all refinements from τ and $e{:}\lfloor\tau\rfloor$ be the standard typing relation for the typed lambda calculus. Then, we define the denotation of types as:

$$
\begin{array}{l}
[\![\{x{:}B \mid p\}]\!] \doteq \{e \mid e{:}B,\ \text{if } e \hookrightarrow^* v \text{ then } p\,[v/x] \hookrightarrow^* \texttt{true}\} \\
[\![x{:}\tau_x \rightarrow \tau]\!] \doteq \{e \mid e{:}\lfloor\tau_x \rightarrow \tau\rfloor, \forall e_x \in [\![\tau_x]\!].\ e\ e_x \in [\![\tau\,[e_x/x]]\!]\}
\end{array}
$$

Constants Each constant c is in the denotation of its type $\mathsf{Ty}(c)$:

$$
\begin{array}{lcl}
\mathsf{Ty}(3) & \doteq & \{v{:}\texttt{int} \mid v = 3\} \\
\mathsf{Ty}(+) & \doteq & x{:}\texttt{int} \rightarrow y{:}\texttt{int} \rightarrow \{v{:}\texttt{int} \mid v = x + y\} \\
\mathsf{Ty}(/) & \doteq & \texttt{int} \rightarrow \{v{:}\texttt{int} \mid v > 0\} \rightarrow \texttt{int} \\
\mathsf{Ty}(\texttt{error}_\tau) & \doteq & \{v{:}\texttt{int} \mid \texttt{false}\} \rightarrow \tau
\end{array}
$$

Thus, if $\mathsf{Ty}(c) \doteq x{:}\tau_x \rightarrow \tau$, then for every value $v \in [\![\tau_x]\!]$, we require that $\delta(c, v) \in [\![\tau\,[v/x]]\!]$. For every value $v \notin [\![\tau_x]\!]$, it suffices to define $\delta(c, v)$ as `crash`, a special untyped value.

Data Constructors The types of data constructor constants are refined with predicates that track the semantics of the *measures* associated with the data type. For example, recall the measure `emp` shown in § 2.5. We encode it as a function `emp` defined as:

$$
\texttt{emp} = \lambda x.\texttt{case}\ \_ = x\ \texttt{of}\ \left\{ \begin{array}{ll} \texttt{NNull} & \rightarrow \texttt{true} \\ \texttt{NCons}\ \_\ \_ & \rightarrow \texttt{false} \end{array} \right\}
$$

and use `emp` to refine the data constructors' types:

$$
\begin{array}{lcl}
\mathsf{Ty}(\texttt{NNull}) & \doteq & \{v{:}\texttt{NList} \mid \texttt{emp}\ v\} \\
\mathsf{Ty}(\texttt{NCons}) & \doteq & \texttt{int} \rightarrow \texttt{NList} \rightarrow \{v{:}\texttt{NList} \mid \neg(\texttt{emp}\ v)\}
\end{array}
$$

We *compose* multiple measures for a type by refining the constructors with the *conjunction* of each measure's refinements.

3.4 Type Checking

Next, we present the type-checking judgments and rules of λ^U.

Environments and Closing Substitutions A *type environment* Γ is a sequence of type bindings $x_1{:}\tau_1, \ldots, x_n{:}\tau_n$. An environment denotes a set of *closing substitutions* θ which are sequences of expression bindings: $x_1 \mapsto e_1, \ldots, x_n \mapsto e_n$ such that:

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

Judgments We use environments to define four kinds of rules: Well-formedness, Subtyping, Implication, and Typing [5, 23]. A judgment $\Gamma \vdash \tau$ states that the refinement type τ is well-formed in the environment Γ. Intuitively, the type τ is well-formed if all the refinements in τ are `bool`-typed in Γ. A judgment $\Gamma \vdash \tau_1 \preceq \tau_2$ states that the type τ_1 is a subtype of τ_2 in the environment Γ. Informally, τ_1 is a subtype of τ_2 if, when the free variables of τ_1 and τ_2 are bound to expressions described by Γ, the denotation of τ_1 is *contained in* the denotation of τ_2. Subtyping of basic types reduces to implication checking. A judgment $\Gamma \vdash p_1 \Rightarrow p_2$ states that the predicate p_1 *implies* the predicate p_2 in the environment Γ. That is, for any closing substitution θ in the denotation of Γ, if $\theta(p_1)$ reduces to `true`, then so does $\theta(p_2)$. A judgment $\Gamma \vdash e : \tau$ states that the expression e has the type τ in the environment Γ. That is, when the free variables in e are bound to expressions described by Γ, the expression e will evaluate to a value described by τ.

Well-Formedness $\boxed{\Gamma \vdash \tau}$

$$\frac{\Gamma, v{:}B \vdash e : \texttt{bool}}{\Gamma \vdash \{v{:}B \mid e\}} \text{ WF-BASE}$$

$$\frac{\Gamma \vdash \tau_x \quad \Gamma, x{:}\tau_x \vdash \tau}{\Gamma \vdash x{:}\tau_x \to \tau} \text{ WF-FUN}$$

Subtyping $\boxed{\Gamma \vdash \tau_1 \preceq \tau_2}$

$$\frac{\Gamma, v : b \vdash e_1 \Rightarrow e_2}{\Gamma \vdash \{v{:}b \mid e_1\} \preceq \{v{:}b \mid e_2\}} \preceq\text{-BASE}$$

$$\frac{\Gamma \vdash \tau'_x \preceq \tau_x \quad \Gamma, x{:}\tau'_x \vdash \tau \preceq \tau'}{\Gamma \vdash x{:}\tau_x \to \tau \preceq x{:}\tau'_x \to \tau'} \preceq\text{-FUN}$$

Implication $\boxed{\Gamma \vdash e_1 \Rightarrow e_2}$

$$\frac{\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \theta(e_1) \hookrightarrow^* \texttt{true} \Rightarrow \theta(e_2) \hookrightarrow^* \texttt{true}}{\Gamma \vdash e_1 \Rightarrow e_2} \text{ T-IMP}$$

Typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau} \text{ T-VAR} \qquad \frac{}{\Gamma \vdash c : \mathsf{Ty}(c)} \text{ T-CON}$$

$$\frac{\Gamma \vdash e : \tau' \quad \Gamma \vdash \tau' \preceq \tau \quad \Gamma \vdash \tau}{\Gamma \vdash e : \tau} \text{ T-SUB}$$

$$\frac{\Gamma, x{:}\tau_x \vdash e : \tau \quad \Gamma \vdash \tau_x}{\Gamma \vdash \lambda x.e : (x{:}\tau_x \to \tau)} \text{ T-FUN}$$

$$\frac{\Gamma \vdash e_1 : (x{:}\tau_x \to \tau) \quad \Gamma \vdash e_2 : \tau_x}{\Gamma \vdash e_1\ e_2 : \tau\,[e_2/x]} \text{ T-APP}$$

$$\frac{\Gamma \vdash e_x : \tau_x \quad \Gamma, x{:}\tau_x \vdash e : \tau \quad \Gamma \vdash \tau}{\Gamma \vdash \texttt{let}\ x = e_x\ \texttt{in}\ e : \tau} \text{ T-LET}$$

$$\frac{\begin{array}{c}\Gamma \vdash e : \{v{:}T \mid e_T\} \qquad \Gamma \vdash \tau \\ \forall i.0 < i \leq \mathsf{Arity}(T).(\\ \mathsf{Ty}(D_{T_i}) = y_1{:}\tau_1 \to \cdots \to y_n{:}\tau_n \to \{v{:}T \mid e_{T_i}\} \\ \Gamma, \overline{y_j{:}\tau_j}, x{:}\{v{:}T \mid e_T \wedge e_{T_i}\} \vdash e_i : \tau)\end{array}}{\Gamma \vdash \texttt{case}\ x = e\ \texttt{of}\ \{D_{T_i}\ \overline{y_j} \to e_i\} : \tau} \text{ T-CASE}$$

Figure 2. Type-checking for λ^U

Soundness Following λ^H [23], we use the (undecidable) T-IMP to show that each step of evaluation preserves typing, and that if an expression is not a value, then it can be further evaluated:

- **Preservation:** If $\emptyset \vdash e : \tau$ and $e \hookrightarrow e'$, then $\vdash e' : \tau$.
- **Progress:** If $\emptyset \vdash e : \tau$ and $e \neq v$, then $e \hookrightarrow e'$.

We combine the above to prove that evaluation preserves typing, and that a well typed term will not `crash`.

Theorem 1. *[Soundness of λ^U]*

- ***Type-Preservation:*** *If $\emptyset \vdash e{:}\tau$ and $e \hookrightarrow^* v$ then $\emptyset \vdash v{:}\tau$.*
- ***Crash-Freedom:*** *If $\emptyset \vdash e{:}\tau$ then $e \not\hookrightarrow^* \texttt{crash}$.*

We prove the above following the overall recipe of [23]. Crash-freedom follows from type-preservation and as `crash` has no type. The Substitution Lemma, in particular, follows from a connection between the typing relation and type denotations:

Lemma 1. *[Denotation Typing] If $\emptyset \vdash e : \tau$ then $e \in [\![\tau]\!]$.*

Label	l	::=	$\downarrow \mid \Downarrow$
Types	τ	::=	$\{v{:}B \mid p\} \mid \{v{:}B^l \mid p\} \mid x{:}\tau \to \tau$

Implication $\boxed{\Gamma \vdash p \Rightarrow p}$

$$\frac{(\!|\Gamma|\!) \Rightarrow (\!|p_1|\!) \Rightarrow (\!|p_2|\!) \text{ is u-valid}}{\Gamma \vdash p_1 \Rightarrow p_2} \text{ D-IMP}$$

Subtyping $\boxed{\Gamma \vdash \tau_1 \preceq \tau_2}$

$$\frac{\Gamma, v : B \vdash p_1 \Rightarrow p_2}{\Gamma \vdash \{v{:}B \mid p_1\} \preceq \{v{:}B \mid p_2\}} \preceq\text{-BASE}$$

Typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{\Gamma \vdash e_1 : (x{:}\tau_x \to \tau) \quad \Gamma \vdash y : \tau_x}{\Gamma \vdash e_1\ y : \tau\,[y/x]} \text{ T-APP}$$

$$\frac{\begin{array}{c}l \notin \{\Downarrow, \downarrow\} \Rightarrow \tau \text{ is Div} \qquad \Gamma \vdash e : \{v{:}T^l \mid e_T\} \qquad \Gamma \vdash \tau \\ \forall i.0 < i \leq \mathsf{Arity}(T).(\\ \mathsf{Ty}(D^i_T) = y_1{:}\tau_1 \to \cdots \to y_n{:}\tau_n \to \{v{:}T \mid e_{T_i}\} \\ \Gamma, \overline{y_j{:}\tau_j}, x{:}\{v{:}T^{\downarrow} \mid e_T \wedge e_{T_i}\} \vdash e_i : \tau)\end{array}}{\Gamma \vdash \texttt{case}\ x = e\ \texttt{of}\ \{D^i_T\ \overline{y_j} \to e_i\} : \tau} \text{ T-CASE}$$

Figure 3. From λ^U to λ^D

Operators	$\oplus$	::=	$+ \mid - \mid \ldots$
Measures	f	::=	$\text{emp} \mid \ldots$
Integers	n	::=	$0 \mid 1 \mid -1 \mid \ldots$
Terms	t	::=	$n \mid x \mid f\,\overline{t} \mid D\,\overline{t} \mid t \oplus t$
		$\mid$	$\texttt{true} \mid \texttt{false}$
Predicates	p	::=	$t = t \mid t < t \mid t \mid p \wedge p \mid \neg p$
Domain	d	::=	$n \mid c_v \mid D\,\overline{d} \mid \texttt{true} \mid \texttt{false}$
Model	σ	::=	$x_1 \mapsto d_1, \ldots, x_n \mapsto d_n$
Lifted Values	$v^{\perp}$	::=	$c \mid \lambda x.e \mid D\,\overline{v^{\perp}} \mid \perp$

Figure 4. Syntax of QF-EUFLIAD

4. Algorithmic Typing: λ^D

While λ^U is sound, it cannot be *implemented* thanks to the undecidable implication checking rule T-IMP (Figure 2). Next, we go from λ^U to λ^D, a core calculus with sound, SMT-based algorithmic type-checking in four steps, summarized in Figure 3. First, we show how to restrict the language of refinements to an SMT-decidable sub-language QF-EUFLIAD (§ 4.1). Second, we *stratify* the types to specify whether their inhabitants may diverge, must reduce to values, or must reduce to finite values (§ 4.2). Third, we show how to *enforce* the stratification by encoding recursion using special fixpoint combinator constants (§ 4.2). Finally, we show how to use QF-EUFLIAD and the stratification to approximate the undecidable T-IMP with a decidable verification condition D-IMP, thereby obtaining the algorithmic system λ^D (§ 4.3).

4.1 Refinement Logic: QF-EUFLIAD

Figure 4 summarizes the syntax of QF-EUFLIAD, the decidable logic of equality, uninterpreted functions, linear arithmetic, and data types [13, 24]. Logical expressions include integers, booleans, variables, function application, data constructors, and linear arith-

metic expressions. A predicate is a (boolean) expression, equality or inequality between terms, or boolean combination of predicates.

Measures & Axioms The only function symbols in QF-EUFLIAD correspond to *measures*, which are inductively defined functions. Each measure definition is a sequence of equations, one per data constructor, where the body of each equation is a term in QF-EUFLIAD. We characterize measures using axioms translated from the measure definition. For example, `emp` from § 2 yields axioms:

$$(\texttt{emp}\ \texttt{NNull}) \qquad \forall x, xs.\neg(\texttt{emp}\ (\texttt{NCons}\ x\ xs))$$

Well-Formedness For a predicate to be well-formed, arithmetic operators should be applied to integer terms, measures should be applied to appropriate arguments (*i.e.* `emp` is applied to `NList`), and equality or inequality to basic (integer or boolean) terms. Furthermore, we require that refinements, and thus measures, always evaluate to a value. We capture these requirements by assigning appropriate types to operators and measure functions, after which the judgment $\emptyset \vdash p : \texttt{bool}$ checks well-formedness.

Assignments Figure 4 defines the elements d of the domain $\mathcal{D}$ of integers, booleans, and data constructors that wrap elements from $\mathcal{D}$. The domain $\mathcal{D}$ also contains a constant c_v for each value v of λ^U that does not otherwise belong in $\mathcal{D}$ (*e.g.* functions or other primitives). An *assignment* σ is a map from variables to $\mathcal{D}$.

Satisfiability & Validity We interpret predicates in the logic over the domain $\mathcal{D}$. We write $\sigma \models p$ (resp. $\sigma \models_D p$) if σ is a model of p where the interpretations of measures respect the measure axioms (resp. measures are uninterpreted). We omit the formal definition for space. A predicate p is *satisfiable* (resp. *u-satisfiable*) if there exists $\sigma \models p$ (resp. $\sigma \models_D p$). A predicate p is *valid* (resp. *u-valid*) if all assignments $\sigma \models p$ (resp. $\sigma \models_D p$). Note that if p is u-valid then p is trivially valid.

Connecting Evaluation and Logic To prove soundness, we need to formally connect the notion of logical models with the evaluation of a refinement to `true`. We do this in several steps, briefly outlined for brevity. First, we introduce a primitive *bottom expression* $\perp$ that can have *any* Div type, but does not evaluate. Second, we define *lifted values* $v^\perp$ (Figure 4), which are values that contain $\perp$. Third, we define *lifted substitutions* $\theta^\perp$, which are mappings from variables to lifted values. Finally, we show how to *embed* a lifted substitution $\theta^\perp$ into a *set of* assignments $(\!|\theta^\perp|\!)$ where, intuitively speaking, each $\perp$ is replaced by some arbitrarily chosen element of $\mathcal{D}$. Now, we can connect evaluation and logical satisfaction:

Theorem 2. *If* $\emptyset \vdash \theta^\perp(p) : \texttt{bool}$*, then*

$$\theta^\perp(p) \hookrightarrow^* \texttt{true} \textit{ iff } \forall \sigma \in (\!|\theta^\perp|\!).\sigma \models p$$

Restricting Refinements to Predicates Our goal is to restrict T-IMP so that only predicates from the decidable logic QF-EUFLIAD (not arbitrary expressions) appear in implications $\Gamma \vdash p_1 \Rightarrow p_2$. Towards this goal, as shown in Figure 3, we restrict the syntax and well-formedness of types to contain only predicates, and we convert the program to ANF after which we can restrict the application rule T-APP to applications to variables, which ensures that refinements remain within the logic after substitution [30].

4.2 Stratified Types

Instead of *explicitly* reasoning about divergence or strictness in the refinement logic (which leads to significant theoretical and practical problems, as discussed in § 8), we choose to reason *implicitly* about divergence within the type system. Thus, the second critical step in our path to λ^D is the stratification of types into those inhabited by potentially diverging terms, terms that only reduce to values, and terms which reduce to finite values. Furthermore, the stratification crucially allows us to prove Theorem 10, which requires that refinements do not diverge (*e.g.* by computing the length of an infinite list) by ensuring that inductively defined measures are only applied to finite values. Next, we describe how we stratify types with labels, and then type the various constants, in particular the fixpoint combinators, to enforce stratification.

Labels We specify stratification using two *labels* for types. The label $\downarrow$ (resp. $\Downarrow$) is assigned to types given to expressions that reduce to a value v (resp. *finite* value, *i.e.* an element of the inductively defined $\mathcal{D}$). Formally,

$$\textbf{\textit{Wnf types}} \quad [\![\{v{:}B^{\downarrow} \mid p\}]\!] \doteq [\![\{v{:}B \mid p\}]\!] \cap \{e \mid e \hookrightarrow^* v\} \quad (8)$$

$$\textbf{\textit{Fin types}} \quad [\![\{v{:}B^{\Downarrow} \mid p\}]\!] \doteq [\![\{v{:}B^{\downarrow} \mid p\}]\!] \cap \{e \mid e \hookrightarrow^* d\} \quad (9)$$

Unlabelled types are assigned to expressions that may diverge. Note that for any B and refinement p we have

$$[\![\{v{:}B^{\Downarrow} \mid p\}]\!] \subseteq [\![\{v{:}B^{\downarrow} \mid p\}]\!] \subseteq [\![\{v{:}B \mid p\}]\!]$$

The first two sets are *equal* for `int` and `bool`, and *unequal* for (lazily) constructed data types T. We need not stratify function types (*i.e.* they are Div types) as binders with function types do not appear inside the VC, and are not applied to measures.

Enforcing Stratification We enforce stratification in two steps. First, the T-CASE rule uses the operational semantics of case-of to type-check each case in an environment where the scrutinee x is assumed to have a Wnf type. All the other rules (not mentioned) in Figure 3 remain the same as in Figure 2. Second, we create stratified variants for the primitive constants and *separate* fixpoint combinator constants for (arbitary, potentially non-terminating) recursion (`fix`) and bounded recursion (`tfix`).

Stratified Primitives First, we restrict the primitive operators whose output types are refined with logical operators, so they are only invoked on finite arguments (so that the corresponding refinements are guaranteed to not diverge).

$$\begin{aligned}
\mathsf{Ty}(\texttt{n}) &\doteq \{v{:}\texttt{int}^{\Downarrow} \mid v = n\} \\
\mathsf{Ty}(=) &\doteq x{:}B^{\Downarrow} \to y{:}B^{\Downarrow} \to \{v{:}\texttt{bool}^{\Downarrow} \mid v \Leftrightarrow x = y\} \\
\mathsf{Ty}(+) &\doteq x{:}\texttt{int}^{\Downarrow} \to y{:}\texttt{int}^{\Downarrow} \to \{v{:}\texttt{int}^{\Downarrow} \mid v = x + y\} \\
\mathsf{Ty}(\wedge) &\doteq x{:}\texttt{bool}^{\Downarrow} \to y{:}\texttt{bool}^{\Downarrow} \to \{v{:}\texttt{bool}^{\Downarrow} \mid v \Leftrightarrow x \wedge y\}
\end{aligned}$$

It is easy to prove that the above primitives respect their stratification labels, *i.e.* belong in the denotations of their types. The only place where divergence enters the picture is through the fixpoint combinators used to encode recursion. For any function or basic type $\tau \doteq \tau_1 \to \ldots \to \tau_n$, we define the *result* to be the type τ_n.

Diverging Fixpoints ($\texttt{fix}_\tau$) For each τ whose result is a Div type, there is a *diverging fixpoint* combinator $\texttt{fix}_\tau$, such that

$$\begin{aligned}
\delta(\texttt{fix}_\tau, f) &\doteq f\ (\texttt{fix}_\tau\ f) \\
\mathsf{Ty}(\texttt{fix}_\tau) &\doteq (\tau \to \tau) \to \tau
\end{aligned}$$

i.e., $\texttt{fix}_\tau$ yields recursive functions of type τ. Of course, $\texttt{fix}_\tau$ belongs in the denotation of its type [29] *only if* the result type is a Div type (and *not* when the result is a Wnf or Fin type). Thus, we restrict diverging fixpoints to functions with Div result types.

Indexed Fixpoints ($\texttt{tfix}^n_\tau$) For each type τ whose result is a Fin type, we have a family of *indexed* fixpoints combinators $\texttt{tfix}^n_\tau$:

$$\begin{aligned}
\delta(\texttt{tfix}^n_\tau, f) &\doteq \lambda m.f\ m\ (\texttt{tfix}^m_\tau\ f) \\
\mathsf{Ty}(\texttt{tfix}^n_\tau) &\doteq (n{:}\texttt{nat}^{\Downarrow} \to \tau_n \to \tau) \to \tau_n \\
\text{where, } \tau_n &\doteq \{v{:}\texttt{nat}^{\Downarrow} \mid v < n\} \to \tau
\end{aligned}$$

τ_n is a *weakened* version of τ that can only be invoked on inputs *smaller* than n. Thus, we enforce termination by requiring that $\texttt{tfix}^n_\tau$ is *only* called with m that are *strictly smaller than* n. As the indices are well-founded `nat`s, evaluation will terminate.

Terminating Fixpoints **($\texttt{tfix}_\tau$)** Finally, we use the indexed combinators to define the *terminating* fixpoint combinator $\texttt{tfix}_\tau$ as:

$$\delta(\texttt{tfix}_\tau, f) \doteq \lambda n.f\ n\ (\texttt{tfix}_\tau^n\ f)$$
$$\mathsf{Ty}(\texttt{tfix}_\tau) \doteq (n{:}\texttt{nat}^{\Downarrow} \to \tau_n \to \tau) \to \texttt{nat}^{\Downarrow} \to \tau$$

Thus, the top-level call to the recursive function requires a $\texttt{nat}^{\Downarrow}$ parameter n that acts as a *starting* index, after which, all "recursive" calls are to combinators with *smaller* indices, ensuring termination.

Example: Factorial Consider the factorial function:

$$\texttt{fac} \doteq \lambda n.\lambda f.\texttt{case}\ \_ = (n = 0)\ \texttt{of}\ \left\{ \begin{array}{l} \texttt{true} \to 1 \\ \_ \to n \times f(n-1) \end{array} \right\}$$

Let $\tau \doteq \texttt{nat}^{\Downarrow}$. We prove termination by typing

$$\emptyset \vdash_D \texttt{tfix}_\tau\ \texttt{fac} : \texttt{nat}^{\Downarrow} \to \tau$$

To understand *why*, note that

$$\begin{aligned} \texttt{tfix}_\tau\ \texttt{fac}\ n &\hookrightarrow^* \texttt{fac}\ n\ (\texttt{tfix}_\tau^n\ \texttt{fac}) \\ &\hookrightarrow^* n \times (\texttt{tfix}_\tau^n\ \texttt{fac}\ (n-1)) \\ &\hookrightarrow^* n \times (\texttt{fac}\ (n-1)\ (\texttt{tfix}_\tau^{n-1}\ \texttt{fac})) \\ &\hookrightarrow^* n \times n - 1 \times (\texttt{tfix}_\tau^{n-1}\ \texttt{fac}\ (n-2)) \\ &\hookrightarrow^* n \times n - 1 \times \ldots \times (\texttt{tfix}_\tau^{1}\ \texttt{fac}\ 0) \\ &\hookrightarrow^* n \times n - 1 \times \ldots \times (\texttt{fac}\ 0\ (\texttt{tfix}_\tau^{0}\ \texttt{fac})) \\ &\hookrightarrow^* n \times n - 1 \times \ldots \times 1 \end{aligned}$$

Soundness of Stratification To formally *prove* that stratification is soundly enforced, it suffices to prove that the Denotation Lemma 1 holds for λ^D. This, in turn, boils down to proving that each (stratified) constant belongs in its type's denotation.

Lemma 2. *[Constant Typing] Every constant* $c \in [\![\mathsf{Ty}(c)]\!]$.

The crucial part of the above is proving that the indexed and terminating fixpoints inhabit their types' denotations.

Theorem 3. *[Fixpoint Typing]*

- $\texttt{fix}_\tau \in [\![\mathsf{Ty}(\texttt{fix}_\tau)]\!]$,
- $\forall n.\texttt{tfix}_\tau^n \in [\![\mathsf{Ty}(\texttt{tfix}_\tau^n)]\!]$,
- $\texttt{tfix}_\tau \in [\![\mathsf{Ty}(\texttt{tfix}_\tau)]\!]$.

4.3 Verification With Stratified Types

We can put the pieces together to obtain an algorithmic implication rule D-IMP instead of the undecidable T-IMP (from Figure 2). Intuitively, each closing substitution θ will correspond to a set of logical assignments $(\!|\theta|\!)$. Thus, we will translate Γ, p_1, and p_2 into logical formulas $(\!|\Gamma|\!)$, $(\!|p_1|\!)$, and $(\!|p_2|\!)$ such that:

- $\theta(p_i) \hookrightarrow^* \texttt{true}$ iff each $\sigma \in (\!|\theta|\!)$ satisfies $(\!|p_i|\!)$, and
- $\theta \in [\![\Gamma]\!]$ iff each $\sigma \in (\!|\theta|\!)$ satisfies $(\!|\Gamma|\!)$.

Translating Refinements & Environments As refinements p are drawn from the restricted logic, they can be trivially translated into formulas, *i.e.* $(\!|p|\!) \doteq p$. To translate environments, recall that $\theta \in [\![\Gamma]\!]$ iff for each $x{:}\tau \in \Gamma$, we have $\theta(x) \in [\![\theta(\tau)]\!]$. Thus,

$$(\!|x_1{:}\tau_1, \ldots, x_n{:}\tau_n|\!) \doteq (\!|x_1{:}\tau_1|\!) \wedge \ldots \wedge (\!|x_n{:}\tau_n|\!)$$

How should we translate a single binding? Since a binding denotes

$$[\![\{x{:}B \mid p\}]\!] \doteq \{e \mid \text{if } e \hookrightarrow^* v \text{ then } p\,[v/x] \hookrightarrow^* \texttt{true}\}$$

a direct translation would require a logical value predicate $\mathsf{Val}(x)$, which we could use to obtain the logical translation

$$(\!|\{x{:}B \mid p\}|\!) \doteq \neg\mathsf{Val}(x) \vee p$$

This translation poses several theoretical and practical problems that preclude the use of existing SMT solvers (as detailed in § 8). However, our stratification guarantees (cf. (8), (9)) that labeled types reduces to values, and so we can simply conservatively translate the Div and labeled (Wnf, Fin) bindings as:

$$(\!|\{x{:}B \mid p\}|\!) \doteq \texttt{true} \qquad (\!|\{x{:}B^l \mid p\}|\!) \doteq p$$

Soundness We prove soundness by showing that the decidable implication D-IMP approximates the undecidable T-IMP.

Theorem 4. *If* $\Gamma \vdash_D p_1 \Rightarrow p_2$ *then* $\Gamma \vdash p_1 \Rightarrow p_2$.

To prove the above, let $VC \doteq (\!|\Gamma|\!) \Rightarrow (\!|p_1|\!) \Rightarrow (\!|p_2|\!)$. First, note that if VC is u-valid then it is valid as the addition of axioms preserves validity. Next, we prove that if the VC is valid then $\Gamma \vdash p_1 \Rightarrow p_2$. This latter fact relies crucially on a notion of *tracking evaluation* which allows us to reduce a closing substitution θ to a lifted substitution $\theta^\perp$, written $\theta \hookrightarrow_\perp^* \theta^\perp$, after which we prove:

Lemma 3. *[Lifting]* $\theta(e) \hookrightarrow^* c$ *iff* $\exists\theta \hookrightarrow_\perp^* \theta^\perp$ *s.t.* $\theta^\perp(e) \hookrightarrow^* c$.

We combine the Lifting Lemma and the equivalence Theorem 10 to prove that the validity of the VC demonstrates the evaluation-based implication $\Gamma \vdash p_1 \Rightarrow p_2$. Finally, the soundness of algorithmic typing follows from Theorems 9 and 1:

Theorem 5. *[Soundness of* λ^D *]*

- ***Type-Preservation:*** *If* $\emptyset \vdash_D e : \tau$ *then* $\emptyset \vdash e : \tau$.
- ***Crash-Freedom:*** *If* $\emptyset \vdash_D e : \tau$ *then* $e \not\hookrightarrow^* \texttt{crash}$.

Uninterpreted Validity Checking We reduce the undecidable implication to checking u-validity of the VC (*i.e.*, validity where measures are uninterpreted). This trivially implies validity of the VC. We use u-validity instead of validity to ensure predictable and efficient checking, as then the VC belongs to QF-EUFLIAD. The elimination of the measure axioms is crucial for efficient and practical verification. The absence of the axioms does not lead to loss of precision in practice; the semantics of the axioms are encoded in the refinement types of the data constructors, and hence already *instantiated* inside (the environment and) the VC during type checking.

5. Implementation: LIQUIDHASKELL

We have implemented λ^D in LIQUIDHASKELL (§ 2). Next, we describe the key steps in the transition from λ^D to Haskell.

5.1 Termination

Haskell's recursive functions of type $\texttt{nat}^{\Downarrow} \to \tau$ are represented, in GHC's Core [34] as $\texttt{let rec}\ f = \lambda n.e$ which is operationally equivalent to $\texttt{let}\ f = \texttt{tfix}_\tau\ (\lambda n.\lambda f.e)$. Given the type of $\texttt{tfix}_\tau$, checking that f has type $\texttt{nat}^{\Downarrow} \to \tau$ reduces to checking e in a *termination-weakened environment* where

$$f : \{v{:}\texttt{nat}^{\Downarrow} \mid v < n\} \to \tau$$

Thus, LIQUIDHASKELL proves termination just as λ^D does: by checking the body in the above environment, where the recursive binder is called with `nat` inputs that are strictly smaller than n.

Default Metric For example, LIQUIDHASKELL proves that

```
fac n = if n == 0 then 1 else n * fac (n-1)
```

has type $\texttt{nat}^{\Downarrow} \to \texttt{nat}^{\Downarrow}$ by typechecking the body of `fac` in a termination-weakened environment $\texttt{fac} : \{v{:}\texttt{nat}^{\Downarrow} \mid v < n\} \to \texttt{nat}^{\Downarrow}$ The recursive call generates the subtyping query:

$$n{:}\{0 \leq n\}, \neg(n = 0) \vdash_D \{\texttt{v} = \texttt{n} - \texttt{1}\} \preceq \{\texttt{0} \leq \texttt{v} \wedge \texttt{v} < \texttt{n}\}$$

Which reduces to the valid VC

$$0 \leq n \wedge \neg(n = 0) \Rightarrow (v = n - 1) \Rightarrow (0 \leq v \wedge v < n)$$

proving that **fac** terminates, in essence because the *first parameter* forms a *well-founded decreasing metric*.

Refinements Enable Termination Consider Euclid's GCD:

```
gcd :: a:Nat -> {v:Nat | v < a} -> Nat
gcd a 0 = a
gcd a b = gcd b (a `mod` b)
```

Here, the first parameter is decreasing, but this requires the fact that the second parameter is smaller than the first and that `mod` returns results smaller than its second parameter. Both facts are easily expressed as refinements, but elude non-extensible checkers [17].

Explicit Termination Metrics The indexed-fixpoint combinator technique is easily extended to cases where some parameter *other* than the first is the well-founded metric. For example, consider:

```
tfac     :: Nat -> n:Nat -> Nat / [n]
tfac x 0 = if n == 0 then x
                     else tfac (n*x) (n-1)
```

We specify that the *last parameter* is decreasing by specifying an explicit termination metric `/ [n]` in the type signature. Type checking now checks the body in an environment where the second argument of `tfac` is weakened, which is proved as before.

Explicit Termination Expressions Sometimes, none of the parameters themselves decrease across recursive calls, but there is some *expression* that forms the decreasing metric. For example consider `range lo hi`, which returns the list of `Int`s from `lo` to `hi`:

```
range :: lo:Nat -> hi:Nat -> [Nat] / [hi-lo]
range lo hi
  | lo < hi   = lo : range (lo + 1) hi
  | otherwise = []
```

Here, neither parameter is decreasing (indeed, the first one is *increasing*) but `hi-lo` decreases across each call. We generalize the explicit metric specification to *expressions* like `hi-lo`. LIQUIDHASKELL *desugars* the expression into a new **nat**-valued *ghost parameter* whose value is always equal to `hi-lo`, that is:

```
range lo hi = go lo hi (hi-lo)
  where
    go lo hi ghost
     | lo<hi = l : go (lo+1) hi (hi-(lo+1))
     | _     = []
```

After which, it proves `go` terminating, by showing that `ghost` is a **nat** that decreases across each recursive call (as in `fac` and `tfac`).

Recursion over Data Types The above strategy generalizes easily to functions that recurse over (finite) data structures like arrays, lists, and trees. In these cases, we simply use *measures* to project the structure onto **nat**, thereby reducing the verification to the previously seen cases. For each user defined type, *e.g.*

```
data L [sz] a = N | C a (L a)
```

we can define a *measure*

```
measure sz  :: L a -> Nat
sz (C x xs) = 1 + (sz xs)
sz N        = 0
```

We prove that `map` terminates using the type:

```
map :: (a -> b) -> xs:L a -> L b / [sz xs]
map f (C x xs) = C (f x) (map f xs)
map f N        = N
```

That is, by simply using `(sz xs)` as the decreasing metric.

Generalized Metrics Over Datatypes Finally, in many functions there is no single argument whose (measure) provably decreases. For example, consider:

```
merge :: xs:_ -> ys:_ -> _ / [sz xs + sz ys]
merge (C x xs) (C y ys)
  | x < y     = x `C` (merge xs  (y `C` ys))
  | otherwise = y `C` (merge (x `C` xs)  ys)
```

from the homonymous sorting routine. Here, neither parameter decreases, but the *sum* of their sizes does. As before LIQUIDHASKELL desugars the decreasing expression into a ghost parameter and thereby proves termination (assuming, of course, that the inputs were finite lists, *i.e.* $\mathtt{L}^{\Downarrow}\ a$.)

Automation: Default Size Measures Structural recursion on the first argument is a common pattern in Haskell code. LIQUIDHASKELL automates termination proofs for this common case, by allowing users to specify a *size measure* for each data type, (*e.g.* `sz` for `L a`). Now, if *no* termination metric is given, by default LIQUIDHASKELL assumes that the *first* argument whose type has an associated size measure decreases. Thus, in the above, we need not specify metrics for `fac` or `gcd` or `map` as the size measure is automatically used to prove termination. This simple heuristic allows us to *automatically* prove 67% of recursive functions terminating.

5.2 Non-termination

By default, LIQUIDHASKELL checks that every function is terminating. We show in § 6 that this is infact the overwhelmingly common case in practice. However, annotating a function as **lazy** deactivates LIQUIDHASKELL's termination check (and marks the result as a Div type). This allows us to check functions that are non-terminating, and allows LIQUIDHASKELL to prove safety properties of programs that manipulate *infinite* data, such as streams, which arise idiomatically with Haskell's lazy semantics. For example, consider the classic `repeat` function:

```
repeat x = x `C` repeat x
```

We cannot use the **tfix** combinators to represent this kind of recursion, and hence, use the non-terminating **fix** combinator instead.

Step 1: Abstract Refinements We can parametrize a datatype with abstract refinements that relate sub-parts of the structure [37]. For example, we parameterize the list type as:

```
data L a <p :: L a -> Prop>
  = N | C a {v: L<p> a | (p v)}
```

which parameterizes the list with a refinement `p` which holds *for each* tail of the list, *i.e.* holds for each of the second arguments to the `C` constructor in each sub-list.

Step 2: Measuring Emptiness Now, we can write a measure that states when a list is *empty*

```
measure emp  :: L a -> Prop
emp N        = true
emp (C x xs) = false
```

As described in § 4, LIQUIDHASKELL translates the abstract refinements and measures into refined types for `N` and `C`.

Step 3: Specification & Verification Finally, we can use the abstract refinements and measures to write a type alias describing a refined version of `L a` representing infinite streams:

```
type Stream a =
  {xs: L <{\v -> not(emp v)}> a | not(emp xs)}
```

We can now type `repeat` as:

```
lazy repeat :: a -> Stream a
repeat x    = x `C` repeat x
```

The **lazy** keyword *deactivates* termination checking, and marks the output as a Div type. Even more interestingly, we can prove safety properties of infinite lists, for example:

```
take             :: Nat -> Stream a -> a
take 0 _         = N
take n (C x xs) = x `C` take (n-1) xs
take _ N         = error "never happens"
```

LIQUIDHASKELL proves, similar to the `head` example from § 2, that we never match a `N` when the input is a `Stream`.

Finite vs. Infinite Lists Thus, the combination of refinements and labels allows our stratified type system to specify and verify whether a list is finite or infinite. Note that: $\mathtt{L}^{\Downarrow}\ a$ represents *finite* lists *i.e.* those produced using the (inductive) terminating fixpoint combinators, $\mathtt{L}^{\downarrow}\ a$ represents (potentially) infinite lists which are guaranteed to reduce to values, *i.e.* non-diverging computations that yield finite or infinite lists, and $\mathtt{L}\ a$ represents computations that may diverge or produce a finite or infinite list.

6. Evaluation

Our goal is to build a practical and effective SMT & refinement type-based verifier for Haskell. We have shown that lazy evaluation requires the verifier to reason about divergence; we have proposed an approach for implicitly reasoning about divergence by eagerly proving termination, thereby optimizing the precision of the verifier. Next, we describe an experimental evaluation of our approach that uses LIQUIDHASKELL to prove termination and functional correctness properties of a suite of widely used Haskell libraries totaling more than 10KLOC. Our evaluation seeks to determine whether our approach is *suitable* for a lazy language (*i.e.* do most Haskell functions terminate?), *precise* enough to capture the termination reasons (*i.e.* is LIQUIDHASKELL able to prove that most functions terminate?), *usable* without placing an unreasonably high burden on the user in the form of explicit termination annotations, and *effective* enough to enable the verification of functional correctness properties. For brevity, we omit a description of the properties other than termination, please see [38] for details.

Implementation LIQUIDHASKELL takes as input: (1) A Haskell *source* file, (2) Refinement type *specifications*, including refined datatype definitions, measures, predicate and type aliases, and function signatures, and (3) Predicate fragments called *qualifiers* which are used to infer refinement types using the abstract interpretation framework of Liquid typing [30]. The verifier returns as output, SAFE or UNSAFE, depending on whether the code meets the specifications or not, and, importantly for debugging the code (or specification!) the inferred types for all sub-expressions.

Benchmarks As benchmarks, we used the following libraries: `GHC.List` and `Data.List`, which together implement many standard list operations, `Data.Set.Splay`, which implements an splay functional set, `Data.Map.Base`, which implements a functional map, `Vector-Algorithms`, which includes a suite of "imperative" array-based sorting algorithms, `Bytestring`, a library for manipulating byte arrays, and `Text`, a library for high-performance Unicode text processing. These benchmarks represent a wide spectrum of idiomatic Haskell codes: the first three are widely used libraries based on recursive data structures, the fourth and fifth perform subtle, low-level arithmetic manipulation of array indices and pointers, and the last is a rich, high-level library with sophisticated application-specific invariants, well outside the scope of even Haskell's expressive type system. Thus, this suite provides a diverse and challenging test-bed for evaluating LIQUIDHASKELL.

Results Table 1 summarizes our experiments, which covered 39 modules totaling 10,202 non-comment lines of source code. The results were collected on a machine with an Intel Xeon X5600 and 32GB of RAM (no benchmark required more than 1GB). Timing data was for runs that performed full verification of safety and functional correctness properties in addition to termination.

Library	LOC	Fun	Rec	Div	Hint	Time
`GHC.List`	310	66	34	5	0	20
`Data.List`	504	97	50	2	6	32
`Data.Map.Base`	1395	180	93	0	12	247
`Data.Set.Splay`	149	35	17	0	7	26
`Bytestring`	3501	569	154	6	73	549
`Text`	3125	493	124	7	44	809
`Vector-Algorithms`	1218	99	31	0	31	196
Total	10202	1539	503	20	173	1880

Table 1. A quantitative evaluation of our experiments. **LOC** is the number of non-comment lines of source code as reported by `sloccount`. **Fun** is the total number of functions in the library. **Rec** is the number of recursive functions. **Div** is the number of functions marked as potentially non-terminating. **Hint** is the number of termination hints, in the form of *termination expressions*, given to LIQUIDHASKELL. **Time** is the time, in seconds, required to run LIQUIDHASKELL.

- *Suitable:* Our approach of eagerly proving termination is in fact, *highly* suitable: of the 503 recursive functions, only 12 functions were *actually* non-terminating (*i.e.* non-inductive). That is, 97.6% of recursive functions are inductively defined.
- *Precise:* Our approach is extremely precise, as refinements provide auxiliary invariants and extensibility that is crucial for proving termination. We successfully *prove* that 96.0% of recursive functions terminate.
- *Usable:* Our approach is highly usable and only places a modest annotation burden on the user. The default metric, namely the first parameter with an associated size measure, suffices to automatically prove 67% of recursive functions terminating. Thus, only 30% require explicit termination metric, totaling about 1.7 witnesses (about 1 line each) per 100 lines of code.
- *Effective:* Our approach is extremely effective at improving the precision of the overall verifier (by allowing the VC to use facts about binders that provably reduce to values.) Without the termination optimization, *i.e.* by only using information for matched-binders (thus in WHNF), LIQUIDHASKELL reports 1,395 unique functional correctness warnings – about 1 per 7 lines. With termination information, this number goes to zero.

7. 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 [6], Agda [26], and Idris [8] typically use *structural* checks where recursion is allowed on sub-terms of ADTs to ensure that *all* terms terminate. We differ in that, since the refinement logic is restricted, we do not require that all functions terminate, and hence, we can prove properties of possibly diverging functions like `collatz` as well as lazy functions like `repeat`. Recent languages like Aura [20] and Zombie [10] allow general recursion, but constrain the logic to a terminating sublanguage, as we do, to avoid reasoning about divergence in the logic. In contrast to us, the above systems crucially assume *call-by-value* semantics to ensure that binders are bound to values, *i.e.* cannot diverge.

Refinement Types are a form of dependent types where invariants are encoded via a combination of types and predicates from a restricted *SMT-decidable* logic [5, 15, 31, 41]. The restriction makes it safe to support arbitrary recursion, which has hitherto never been a problem for refinement types. However, we show that this is because all the above systems implicitly assume that all free variables are bound to values, which is only guaranteed under CBV and, as we have seen, leads to unsoundness under lazy evaluation.

Tracking Divergent Computations The notion of type stratification to track potentially diverging computations dates to at least [11] which uses $\bar{\tau}$ to encode diverging terms, and types `fix` as $(\bar{\tau} \rightarrow \bar{\tau}) \rightarrow \bar{\tau}$). More recently, [9] tracks diverging computations within a *partiality monad*. Unlike the above, we use refinements to obtain terminating fixpoints (`tfix`), which let us prove the vast majority (of sub-expressions) in real world libraries as non-diverging, avoiding the restructuring that would be required by the partiality monad.

Termination Analyses Various authors have proposed techniques to verify termination of recursive functions, either using the "size-change principle" [21, 32], or by annotating types with size indices and verifying that the arguments of recursive calls have smaller indices [3, 19]. Our use of refinements to encode terminating fixpoints is most closely related to [40], but this work also crucially assumes CBV semantics for soundness.

AProVE [17] implements a powerful, fully-automatic termination analysis for Haskell based on term-rewriting. While we could use an external analysis like AProVE, we have found that encoding the termination proof via refinements provided advantages that are crucial in large, real-world code bases. Specifically, refinements let us (1) prove termination over a subset (not all) of inputs; many functions (*e.g.* `fac`) terminate only on `Nat` inputs and not all `Int` s, (2) encode pre-conditions, post-conditions, and auxiliary invariants that are essential for proving termination, (*e.g.* `gcd`), (3) easily specify non-standard decreasing metrics and prove termination, (*e.g.* `range`). In each case, the code could be (significantly) *rewritten* to be amenable to AProVE but this defeats the purpose of an automatic checker. Finally, none of the above analyses have been empirically evaluated on large and complex real-world libraries.

Static Contract Checkers like ESCJava [16] are a classical way of verifying correctness through assertions and pre- and post-conditions. Side-effects like modifications of global variables are a well known issue for static checkers for imperative languages; the standard approach is to use an effect analysis to determine the "modifies clause" *i.e.* the set of globals modified by a procedure. Similarly, one can view our approach as implicitly computing the non-termination effects. [42] describes a static contract checker for Haskell that uses symbolic execution to unroll procedures upto some fixed depth, yielding weaker "bounded" soundness guarantees. Similarly, Zeno [33] is an automatic Haskell prover that combines unrolling with heuristics for rewriting and proof-search. Based on rewriting, it is sound but "Zeno might loop forever" when faced with non-termination. Finally, the Halo [39] contract checker encodes Haskell programs into first-order logic by directly modeling the code's denotational semantics, again, requiring heuristics for instantiating axioms describing functions' behavior. Unlike any of the above, our type-based approach does not rely on heuristics for unrolling recursive procedures, or instantiating axioms. Instead we are based on decidable SMT validity checking and abstract interpretation [30] which makes the tool predictable and the overall workflow scale to the verification of large, real-world code bases.

8. Conclusions & Future Work

Our goal is to use the recent advances in SMT solving to build automated refinement type-based verifiers for Haskell. In this paper, we have made the following advances towards the goal. First, we demonstrated how the classical technique for generating VCs from refinement subtyping queries is unsound under lazy evaluation. Second, we have presented a solution that addresses the unsoundness by stratifying types into those that are inhabited by terms that may diverge, those that must reduce to Haskell values, and those that must reduce to finite values, and have shown how refinement types may themselves be used to soundly verify the stratification. Third, we have developed an implementation of our technique in LIQUIDHASKELL and have evaluated the tool on a large corpus comprising 10KLOC of widely used Haskell libraries. Our experiments empirically demonstrate the practical effectiveness of our approach: using refinement types, we were able to prove 96% of recursive functions as terminating, and to crucially use this information to prove a variety of functional correctness properties.

Limitations While our approach is demonstrably effective *in practice*, it relies critically on proving termination, which, while independently useful, is not wholly satisfying *in theory*, as adding divergence shouldn't *break* a safety proofs. Our system can prove a program safe, but if the program is modified by making some functions non-deterministically diverge, then, since we rely on termination, we may no longer be able to prove safety. Thus, in future work, it would be valuable to explore *other* ways to reconcile laziness and refinement typing. We outline some routes and the challenging obstacles along them.

A. Convert Lazy To Eager Evaluation One alternative might be to translate the program from lazy to eager evaluation, for example, to replace every (thunk) e with an abstraction $\lambda().e$, and every use of a lazy value x with an application $x\ ()$. After this, we could simply assume eager evaluation, and so the usual refinement type systems could be used to verify Haskell. Alas, no. While sound, this translation doesn't solve the problem of reasoning about divergence. A dependent function type $x{:}\texttt{int} \rightarrow \{v{:}\texttt{int} \mid v > x\}$ would be transformed to $x{:}(() \rightarrow \texttt{int}) \rightarrow \{v{:}\texttt{int} \mid v > x\ ()\}$ The transformed type is problematic as it uses arbitrary function applications in the refinement logic! The type is only sensible if $x\ ()$ provably reduces to a value, bringing us back to square one.

B. Explicit Reasoning about Divergence Another alternative is to enrich the refinement logic with a *value predicate* $\mathsf{Val}(x)$ that is true when "x is a value" and use the SMT solver to *explicitly* reason about divergence. (Note that $\mathsf{Val}(x)$ is equivalent to introducing a $\perp$ constant denoting divergence, and writing $(x \neq \perp)$.) Unfortunately, this $\mathsf{Val}(x)$ predicate takes the VCs outside the scope of the standard efficiently decidable logics supported by SMT solvers. To see why, recall the subtyping query (1) from `good` in § 2. With explicit value predicates, this subtyping reduces to the VC:

$$\begin{array}{l}(\mathsf{Val}(x) \Rightarrow x \geq 0) \\ (\mathsf{Val}(y) \Rightarrow y \geq 0)\end{array} \Rightarrow (v = y + 1) \Rightarrow (v > 0) \qquad (10)$$

To prove the above valid, we require the knowledge that $(v = y{+}1)$ implies that y is a value, *i.e.* that $\mathsf{Val}(y)$ holds. This fact, while obvious to a *human* reader, is outside the decidable theories of linear arithmetic of the existing SMT solvers. Thus, existing solvers would be unable to prove (10) valid, causing us to reject `good`.

Possible Fix: Explicit Reasoning With Axioms? One possible fix for the above would be to specify a collection of *axioms* that characterize how the value predicate behaves with respect to the other theory operators. For example, we might specify axioms like:

$$\forall x, y, z.(x = y + z) \ \Rightarrow\ (\mathsf{Val}(x) \wedge \mathsf{Val}(y) \wedge \mathsf{Val}(z))$$
$$\forall x, y.(x < y) \ \Rightarrow\ (\mathsf{Val}(x) \wedge \mathsf{Val}(y))$$

etc.. However, this is a non-solution for several reasons. First, it is not clear what a complete set of axioms is. Second, there is the well known loss of predictable checking that arises when using axioms, as one must rely on various brittle, syntactic matching and instantiation heuristics [14]. It is unclear how well these heuristics will work with the sophisticated linear programming-based algorithms used to decide arithmetic theories. Thus, proper support for value predicates could require significant changes to existing decision procedures, making it impossible to use existing SMT solvers.

Possible Fix: Explicit Reasoning With Types? Another possible fix would be to encode the behavior of the value predicates within the refinement types for different operators, after which the predicate

itself could be treated as an *uninterpreted function* in the refinement logic [7]. For instance, we could type the primitives:

```
(+) :: x:Int -> y:Int
    -> {v | v  =  x + y && Val x && Val y}
(<) :: x:Int -> y:Int
    -> {v | v <=> x < y && Val x && Val y}
```

While this approach requires *no* changes to the SMT machinery, it makes specifications complex and verbose. We cannot just add the value predicates to the primitives' specifications. Consider

```
choose b x y = if b then x+1 else y+2
```

To reason about the output of `choose` we must type it as:

```
choose :: Bool -> x:Int -> y:Int
       -> {v|(v > x && Val x)||(v > y && Val y)}
```

Thus, the value predicates will pervasively clutter all signatures with strictness information, making the system unpleasant to use.

Divergence Requires 3-Valued Logic Finally, for either "fix", the value predicate poses a model-theoretic problem: what is the meaning of $\mathsf{Val}(x)$? One sensible approach is to extend the universe with a family of *distinct* $\perp$ constants, such that $\mathsf{Val}(\perp)$ is false. These constants lead inevitably into a three-valued logic (in order to give meaning to formulas like $\perp = \perp$). Thus, even if we were to find a way to reason with the value predicate via axioms or types, we would have to ensure that we properly handled the 3-valued logic within existing 2-valued SMT solvers.

Future Work Thus, in future work it would be worthwhile to address the above technical and usability problems to enable explicit reasoning with the value predicate. This explicit system would be *more expressive* than our stratified approach, *e.g.* would let us check `let x = collatz 10 in 12 `div` x+1` by encoding strictness inside the logic. Nevertheless, we suspect such a verifier would use stratification to eliminate the value predicate in the common case. At any rate, until these hurdles are crossed, we can take comfort in stratified refinement types and can just *eagerly* use termination to prove safety for *lazy* languages.

References

[1] Collatz Conjecture. http://en.wikipedia.org/wiki/Collatz_conjecture.

[2] L. Augustsson. Cayenne - a language with dependent types. In *ICFP*, 1998.

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

[4] J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In *ESOP*, 2011.

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

[6] Y. Bertot and P. Castéran. *Coq'Art: The Calculus of Inductive Constructions*. Springer Verlag, 2004.

[7] A. Bradley and Z. Manna. *The Calculus of Computation: Decision Procedures With Application To Verification*. Springer-Verlag, 2007.

[8] E. Brady. Idris: general purpose programming with dependent types. In *PLPV*, 2013.

[9] V. Capretta. General recursion via coinductive types. *Logical Methods in Computer Science*, 2005.

[10] C. Casinghino, V. Sjöberg, and S. Weirich. Combining proofs and programs in a dependently typed language. In *POPL*, 2014.

[11] R. L. Constable and S. F. Smith. Partial objects in constructive type theory. In *LICS*, 1987.

[12] D. Coutts, R. Leshchinskiy, and D. Stewart. Stream fusion: from lists to streams to nothing at all. In *ICFP*, 2007.

[13] L. de Moura and N. Bjørner. Z3: An efficient SMT solver. 2008.

[14] David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

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

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

[17] J. Giesl, M. Raffelsieper, P. Schneider-Kamp, S. Swiderski, and R. Thiemann. Automated termination proofs for Haskell by term rewriting. *TPLS*, 2011.

[18] C. A. R. Hoare. Procedures and parameters: An axiomatic approach. In *Symposium on Semantics of Algorithmic Languages*. 1971.

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

[20] L. Jia, J. A. Vaughan, K. Mazurak, J. Zhao, L. Zarko, J. Schorr, and S. Zdancewic. Aura: a programming language for authorization and audit. In *ICFP*, 2008.

[21] N. D. Jones and N. Bohr. Termination analysis of the untyped lambda-calculus. In *RTA*, 2004.

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

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

[24] G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.

[25] T. Nipkow. Hoare logics for recursive procedures and unbounded nondeterminism. In *CSL*, 2002.

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

[27] 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.

[28] B. O'Sullivan, S. Marlow, D. Roundy, and D. Stewart. bytestring-0.9.2.1. http://hackage.haskell.org/package/bytestring-0.9.2.1.

[29] S. R. Della Rocca and L. Paolini. *The Parametric Lambda Calculus, A Metamodel for Computation*. 2004.

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

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

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

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

[34] M. Sulzmann, M. M. T. Chakravarty, S. L. Peyton-Jones, and K. Donnelly. System F with type equality coercions. In *TLDI*, 2007.

[35] N. Swamy, J. Chen, C. Fournet, P-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, 2011.

[36] A. M. Turing. On computable numbers, with an application to the eintscheidungsproblem. In *LMS*, 1936.

[37] N. Vazou, P. Rondon, and R. Jhala. Abstract refinement types. In *ESOP*, 2013.

[38] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. Peyton-Jones. http://goto.ucsd.edu/~nvazou/haskell-refinements-techrep.pdf.

[39] D. Vytiniotis, S.L. Peyton-Jones, K. Claessen, and D. Rosén. Halo: haskell to logic through denotational semantics. In *POPL*, 2013.

[40] H. Xi. Dependent types for program termination verification. In *LICS*, 2001.

[41] H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.

[42] D. N. Xu, S. L. Peyton-Jones, and K. Claessen. Static contract checking for haskell. In *POPL*, 2009.

Constants	c	::=	$0, 1, -1, \ldots \mid \texttt{true}, \texttt{false}$
		$\mid$	$+, -, \ldots \mid =, <, \ldots \mid \texttt{crash}$
Values	v	::=	$c \mid \lambda x.e \mid D\,\overline{e}$
Expressions	e	::=	$\bot \mid v \mid x \mid e\,e \mid \texttt{let}\ x = e\ \texttt{in}\ e$
		$\mid$	$\texttt{case}\ x = e\ \texttt{of}\ \{D\,\overline{x} \to e\}$
Basic Types	B	::=	$\texttt{int} \mid \texttt{bool} \mid T$
Label	l	::=	$\downarrow \mid \Downarrow$
Types	τ	::=	$\{v{:}B \mid e\} \mid \{v{:}B^l \mid e\} \mid x{:}\tau \to \tau$
Contexts	C	::=	$\bullet \mid C\,e \mid c\,C \mid D\,\overline{e}\,C\,\overline{e}$
		$\mid$	$\texttt{case}\ x = C\ \texttt{of}\ \{D\,\overline{y} \to e\}$

Figure 5. λ^U: Syntax

A. Declarative Typing: λ^U

A.1 Definitions

To simplify the metatheory we extend λ^U so that

- Supports stratified types, and
- explicitly contains $\bot$, a primitive that has any type, but does not evaluate.

Then, we define the function $\lfloor \bullet \rfloor$ that erases the refinements in types and environments:

$$\lfloor \{v{:}B^l \mid e\} \rfloor = B^l \qquad \lfloor \emptyset \rfloor = \emptyset$$
$$\lfloor x{:}\tau_x \to \tau \rfloor = \lfloor \tau_x \rfloor \to \lfloor \tau \rfloor \qquad \lfloor x{:}\tau, \Gamma \rfloor = x{:}\lfloor \tau \rfloor, \lfloor \Gamma \rfloor$$

and variable substitution on types:

$$(\{v{:}B^l \mid e\})\,[e_y/y] = \{v{:}B^l \mid e\,[e_y/y]\}$$
$$(x{:}\tau_x \to \tau)\,[e_y/y] = x{:}(\tau_x\,[e_y/y]) \to (\tau\,[e_y/y])$$

We extend the typing rules with another rule that types $\bot$ with **any** type getting the rules as defined in Figure 6.

We define the denotations of types by combining the denotations of stratified types:

Definition 1. *[Type Denotations]*

$$[\![\{x{:}B \mid p\}]\!] \doteq \{e \mid \emptyset \vdash_B e{:}B,\ \textit{if}\ e \hookrightarrow^* v\ \textit{then}\ p\,[v/x] \hookrightarrow^* \texttt{true}\}$$
$$[\![\{v{:}B^{\downarrow} \mid p\}]\!] \doteq [\![\{v{:}B \mid p\}]\!] \cap \{e \mid e \hookrightarrow^* v\}$$
$$[\![\{v{:}B^{\Downarrow} \mid p\}]\!] \doteq [\![\{v{:}B^{\downarrow} \mid p\}]\!] \cap \{e \mid e \hookrightarrow^* d\}$$
$$[\![x{:}\tau_x \to \tau]\!] \doteq \{e \mid \emptyset \vdash_B e{:}\lfloor \tau_x \to \tau \rfloor, \forall e_x \in [\![\tau_x]\!].\ e\ e_x \in [\![\tau\,[e_x/x]]\!]\}$$

Finally, we define the constraints that should be satisfied by constants:

Definition 2. *[Constants] For every basic type T there are exactly $n = \mathsf{Arity}(T)$ data contractors with result type T, namely $\{D^i_T | 0 < i \leq n\}$.*
$\texttt{crash}$ is an untyped constant. For each constant $c \neq \texttt{crash}$

1. $\emptyset \vdash c : \mathsf{Ty}(c)$ *and* $\vdash \mathsf{Ty}(c)$
2. *If* $\mathsf{Ty}(c) = x{:}\tau_x \to \tau$, *then for each* v, $\delta(c, v)$ *is defined and if* $\emptyset \vdash v : \tau_x$ *then* $\vdash [\![c]\!](v) \in \tau\,[v/x]$, *otherwise* $[\![c]\!](v) = \texttt{crash}$.
3. *If* $\mathsf{Ty}(c) = \{v{:}B^l \mid e\}$, *then* $c \in [\![\mathsf{Ty}(c)]\!]$ *and* $\forall c', c' \neq c.c' \notin [\![\mathsf{Ty}(c)]\!]$
4. *If* $\mathsf{Ty}(D^i_T) = x_1{:}\tau_1 \to \ldots x_n{:}\tau_n \to \tau$, *then* τ_i *are unrefined and for every* e_i *with* $0 < i \leq n$, *such that* $\emptyset \vdash e_i : \tau_i$, $D^i_T\ \overline{e_i} \in [\![\tau\,[e_i/x_i]]\!]$.

Well-Formed Types $\boxed{\Gamma \vdash \tau}$

$$\frac{\Gamma, v{:}B \vdash e : \texttt{bool}}{\Gamma \vdash \{v{:}B \mid e\}}\ \text{WF-BASE}$$

$$\frac{\Gamma \vdash \tau_x \qquad \Gamma, x{:}\tau_x \vdash \tau}{\Gamma \vdash x{:}\tau_x \to \tau}\ \text{WF-FUN}$$

Well-Formed Environments $\boxed{\vdash \Gamma}$

$$\frac{}{\vdash \emptyset} \qquad \frac{\Gamma \vdash \tau_x \qquad \vdash \Gamma}{\vdash x \mapsto \tau_x, \Gamma}$$

Subtyping $\boxed{\Gamma \vdash \tau_1 \preceq \tau_2}$

$$\frac{\Gamma, v : b \vdash e_1 \Rightarrow e_2}{\Gamma \vdash \{v{:}b \mid e_1\} \preceq \{v{:}b \mid e_2\}}\ \preceq\text{-BASE}$$

$$\frac{\Gamma \vdash \tau'_x \preceq \tau_x \qquad \Gamma, x{:}\tau'_x \vdash \tau \preceq \tau'}{\Gamma \vdash x{:}\tau_x \to \tau \preceq x{:}\tau'_x \to \tau'}\ \preceq\text{-FUN}$$

Implication $\boxed{\Gamma \vdash e_1 \Rightarrow e_2}$

$$\frac{\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \theta(e_1) \hookrightarrow^* \texttt{true} \Rightarrow \theta(e_2) \hookrightarrow^* \texttt{true}}{\Gamma \vdash e_1 \Rightarrow e_2}\ \text{T-IMP}$$

Typing $\boxed{\Gamma \vdash e : \tau}$

$$\frac{}{\Gamma \vdash \bot{:}\ \{v{:}B \mid e\}}\ \text{T-BOT}$$

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash x : \tau}\ \text{T-VAR} \qquad \frac{}{\Gamma \vdash c : \mathsf{Ty}(c)}\ \text{T-CON}$$

$$\frac{\Gamma \vdash e : \tau' \qquad \Gamma \vdash \tau' \preceq \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash e : \tau}\ \text{T-SUB}$$

$$\frac{\Gamma, x{:}\tau_x \vdash e : \tau \qquad \Gamma \vdash \tau_x}{\Gamma \vdash \lambda x.e : (x{:}\tau_x \to \tau)}\ \text{T-FUN}$$

$$\frac{\Gamma \vdash e_1 : (x{:}\tau_x \to \tau) \qquad \Gamma \vdash e_2 : \tau_x}{\Gamma \vdash e_1\ e_2 : \tau\,[e_2/x]}\ \text{T-APP}$$

$$\frac{\Gamma \vdash e_x : \tau_x \qquad \Gamma, x{:}\tau_x \vdash e : \tau \qquad \Gamma \vdash \tau}{\Gamma \vdash \texttt{let}\ x = e_x\ \texttt{in}\ e : \tau}\ \text{T-LET}$$

$$\frac{\begin{array}{c} l \notin \{\Downarrow, \downarrow\} \Rightarrow \tau \text{ is Div} \qquad \Gamma \vdash e : \{v{:}T^l \mid e_T\} \qquad \Gamma \vdash \tau \\ \forall i.0 < i \leq \mathsf{Arity}(T).(\\ \mathsf{Ty}(D^i_T) = y_1{:}\tau_1 \to \cdots \to y_n{:}\tau_n \to \{v{:}T \mid e_{T_i}\} \\ \Gamma, \overline{y_j{:}\tau_j}, x{:}\{v{:}T^{\downarrow} \mid e_T \wedge e_{T_i}\} \vdash e_i : \tau) \end{array}}{\Gamma \vdash \texttt{case}\ x = e\ \texttt{of}\ \{D^i_T\ \overline{y_j} \to e_i\} : \tau}\ \text{T-CASE}$$

Figure 6. Type-checking for λ^U

A.2 Denotational Typing

We define denotational typing as follows:

$$\Gamma \vdash e \in \tau \doteq \forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \theta\ e \in [\![\theta\ \tau]\!]$$
$$\Gamma \vdash \tau_1 \subseteq \tau_2 \doteq \forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow [\![\theta\ \tau_1]\!] \subseteq [\![\theta\ \tau_2]\!]$$

And prove that syntactic typing implies denotational typing, *i.e.* a general version of Lemma 1 of the paper.

Lemma 4. *[Denotation Typing]*

1. *If* $\Gamma \vdash \tau_1 \preceq \tau_2$ *then* $\Gamma \vdash \tau_1 \subseteq \tau_2$.
2. *If* $\Gamma \vdash e : \tau$ *then* $\Gamma \vdash e \in \tau$.

Proof. Helping Lemma:

Lemma 5. *If $e \hookrightarrow^* e'$ then $e' \in [\![\tau]\!]$ iff $e \in [\![\tau]\!]$.*

Proof Sketch: Since the validity of $e \in [\![\tau]\!]$ depends on the evaluated e, the if direction is evident. The only if direction follows from the deterministic operational semantics. □

1. Assume $\Gamma \vdash \tau_1 \preceq \tau_2$. We will prove it by induction on the derivation tree:
   - $\preceq$-BASE. We have
     $$\Gamma \vdash \{v{:}B^l \mid e_1\} \preceq \{v{:}B^l \mid e_2\}$$
     By inversion we get
     $$\Gamma, v{:}B \vdash e_1 \Rightarrow e_2$$
     By inversion of T-IMP we have
     $$\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \theta(e_1) \hookrightarrow^* \texttt{true} \Rightarrow \theta(e_2) \hookrightarrow^* \texttt{true}\ (1)$$
     We want to prove
     $$\Gamma \vdash \{v{:}B^l \mid e_1\} \subseteq \{v{:}B^l \mid e_2\}$$
     Equivalently
     $$\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow [\![\theta\ \{v{:}B^l \mid e_1\}]\!] \subseteq [\![\theta\ \{v{:}B^l \mid e_2\}]\!]$$
     Since the labels are the same it suffices to prove that
     $$\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \{e \mid\vdash e : B \wedge \theta(e_1\,[e/v]) \hookrightarrow^* \texttt{true}\} \subseteq \{e \mid\vdash e : B \wedge \theta(e_2\,[e/v]) \hookrightarrow^* \texttt{true}\}$$
     Since $e \in [\![B]\!]$, we have $\Gamma, v{:}B \vdash \theta, [e/v]$. So, from (1) for $\theta := \theta, [e/v]$ we have
     $$\theta(e_1\,[e/v]) \hookrightarrow^* \texttt{true} \Rightarrow \theta(e_2\,[e/v]) \hookrightarrow^* \texttt{true}$$
   - $\preceq$-FUN Assume
     $$\Gamma \vdash x{:}\tau_x \to \tau \preceq x{:}\tau'_x \to \tau'$$
     By inversion we have
     $$\Gamma \vdash \tau'_x \preceq \tau_x \qquad \Gamma, x{:}\tau'_x \vdash \tau \preceq \tau'$$
     By IH
     $$\Gamma \vdash \tau'_x \subseteq \tau_x\ (1) \qquad \Gamma, x{:}\tau'_x \vdash \tau \subseteq \tau'\ (2)$$
     We want to show that
     $$\Gamma \vdash x{:}\tau_x \to \tau \subseteq x{:}\tau'_x \to \tau'$$
     Equivalently
     $$\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow [\![\theta(x{:}\tau_x \to \tau)]\!] \subseteq [\![\theta(x{:}\tau'_x \to \tau')]\!]$$
     Equivalently
     $$\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \{e \mid\vdash e : \lfloor\tau_x\rfloor \to \lfloor\tau\rfloor \wedge \forall e_x \in [\![\theta(\tau_x)]\!].\ e\ e_x \in [\![\theta(\tau\,[e_x/x])]\!]\} \subseteq \{e \mid\vdash e : \lfloor\tau'_x\rfloor \to \lfloor\tau'\rfloor \wedge \forall e_x \in [\![\theta(\tau'_x)]\!].\ e\ e_x \in [\![\theta(\tau'\,[e_x/x])]\!]\}$$
     The above holds, as for any e, e_x if $e_x \in [\![\theta(\tau'_x)]\!]$ then by (1) $e_x \in [\![\theta(\tau_x)]\!]$. Also, by (2) if $e\ e_x \in [\![\theta(\tau\,[e_x/x])]\!]$ then $e\ e_x \in [\![\theta(\tau'\,[e_x/x])]\!]$.
2. Assume $\Gamma \vdash e : \tau$. We will prove it by induction on the derivation tree.
   - T-VAR Assume $\Gamma \vdash e : \tau$ where $e \equiv x$. By inversion we have
     $$(x, \tau) \in \Gamma$$
     We need to show that
     $$\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \theta(x) \in [\![\theta(\tau)]\!]$$
     Which holds by the definition of well-formed substitutions.
   - T-CON. Assume $\Gamma \vdash e : \tau$ where $e \equiv c$ and $\tau \equiv \mathsf{Ty}(c)$. Then $\Gamma \vdash e \in \tau$ holds by Definition 2.
   - T-SUB Assume $\Gamma \vdash e : \tau$. By inversion
     $$\Gamma \vdash e : \tau'\ (1) \qquad \Gamma \vdash \tau' \preceq \tau\ (2) \qquad \Gamma \vdash \tau\ (3)$$
     By IH on (1) we have
     $$\Gamma \vdash e \in \tau'\ (4)$$
     By 1 on (2)
     $$\Gamma \vdash \tau' \subseteq \tau\ (5)$$
     By (4) and (5) we get
     $$\Gamma \vdash e \in \tau$$
   - T-FUN Assume $\Gamma \vdash e : \tau$, where $e \equiv \lambda x.e'$ and $\tau \equiv x{:}\tau'_x \to \tau'$. By inversion we get
     $$\Gamma, x{:}\tau'_x \vdash e' : \tau'\ (1) \qquad \Gamma \vdash \tau'_x\ (2)$$
     By IH on (1) we have
     $$\Gamma, x{:}\tau'_x \vdash e' \in \tau'\ (3)$$
     Equivalently
     $$\forall \theta.(\theta\,[e_x/x]) \in [\![(\Gamma, x{:}\tau'_x)]\!] \Rightarrow (\theta\,[e_x/x])(e') \in [\![(\theta\,[e_x/x])(\tau')]\!]$$
     Or
     $$\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \forall e_x.e_x \in [\![\tau'_x]\!] \Rightarrow \theta(e\ e_x) \in [\![\theta(\tau'\,[e_x/x])]\!]$$
     Moreover, $\vdash_B e{:}\lfloor\tau'_x\rfloor \to \lfloor\tau\rfloor$. So,
     $$\forall \theta.\theta \in [\![\Gamma]\!].\theta(e) \in [\![\theta(\tau)]\!]$$
     Or,
     $$\Gamma \vdash e \in \tau$$
   - T-APP. Assume $\Gamma \vdash e : \tau$, where $e \equiv e_1\ e_2$ and $\tau \equiv \tau'\,[e_2/x]$. By inversion:
     $$\Gamma \vdash e_1 : (x{:}\tau'_x \to \tau')\ (1) \qquad \Gamma \vdash e_2 : \tau'_x\ (2)$$
     By IH we get
     $$\Gamma \vdash e_1 \in (x{:}\tau'_x \to \tau')\ (3) \qquad \Gamma \vdash e_2 \in \tau'_x\ (4)$$
     So
     $$\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \forall e_x \in [\![\theta(\tau'_x)]\!] \Rightarrow (\theta(e_1))\ e_x \in [\![\theta(\tau'\,[e_x/x])]\!]\ (5)$$
     and
     $$\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \theta(e_2) \in [\![\theta(\tau'_x)]\!]\ (6)$$
     From (5) and (6), we get
     $$\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \theta\ e \in [\![\theta(\tau)]\!]$$
     Or
     $$\Gamma \vdash e \in \tau$$
   - T-LET. Assume $\Gamma \vdash e : \tau$, where $e \equiv \texttt{let}\ x = e_x\ \texttt{in}\ e'$. By inversion:
     $$\Gamma \vdash e_x : \tau_x\ (1) \qquad \Gamma, x{:}\tau_x \vdash e' : \tau\ (2) \qquad \Gamma \vdash \tau\ (3)$$
     By IH we get
     $$\Gamma \vdash e_x \in \tau_x\ (4) \qquad \Gamma, x{:}\tau_x \vdash e' \in \tau\ (5)$$
     By (5)
     $$\forall \theta'.\theta' \in [\![\Gamma, x : \tau_x]\!] \Rightarrow \theta'(e') \in [\![\theta'(\tau)]\!]\ (6)$$
     By (4),
     $$\theta \in [\![\Gamma]\!] \Rightarrow \theta\,[e_x/x] \in [\![\Gamma, x : \tau_x]\!]\ (7)$$
     From (6), (7) and (3), we get
     $$\forall \theta.\theta \in [\![\Gamma]\!] \Rightarrow \theta(e'\,[e_x/x]) \in [\![\theta(\tau)]\!]$$

By Lemma 5, we get

$$\forall\theta.\theta \in [\![\Gamma]\!] \Rightarrow \theta(e) \in [\![\theta(\tau)]\!]$$

So,

$$\Gamma \vdash e \in \tau$$

- T-BOT Assume $\Gamma \vdash e : \tau$, where $e \equiv \bot$ and $\tau \equiv \{v{:}B \mid p\}$. Since $\bot$ does not evaluate,

$$\forall\theta.\theta \in [\![\Gamma]\!] \Rightarrow \theta(e) \in [\![\theta(\tau)]\!]$$

So,

$$\Gamma \vdash e \in \tau$$

- T-CASE Assume $\Gamma \vdash e : \tau$, where $e' \equiv \texttt{case}\ x = e\ \texttt{of}\ \{D_T^i\ \overline{y} \to e_i\}$. By inversion

$$l \notin \{\Downarrow, \downarrow\} \Rightarrow \tau \text{ is Div } (1) \qquad \Gamma \vdash e : \{v{:}T^l \mid e_T\}\ (2) \qquad \Gamma \vdash \tau\ (3)$$

$$\forall i.0 < i \leq \mathsf{Arity}(T)\{$$

$$\mathsf{Ty}(D_T^i) = y_1{:}\tau_1 \to \cdots \to y_n{:}\tau_n \to \{v{:}T \mid e_{T_i}\}\ (4)$$

$$\Gamma, \overline{y_i{:}\tau_i}, x{:}\{v{:}T^{\downarrow} \mid e_T \wedge e_{T_i}\} \vdash e_i : \tau\ (5)\}$$

By IH on (2) we get

$$\Gamma \vdash e' \in \{v{:}T^l \mid e_T\}\ (6)$$

We fix a θ such that $\theta \in [\![\Gamma]\!]$ We split cases on whether $\theta(e')$ evaluates to a WNF or not:

- If $\theta(e') \hookrightarrow^* v$. By (6), for some i such that $0 < i \leq \mathsf{Arity}(T)$, $\theta(e') \hookrightarrow^* D_T^i\ \overline{e_j}$.
  By IH on (4) and the Definition 2

$$\Gamma \vdash e_i\,[e_j/y_i]\,[e'/x] \in \tau$$

  Finally, by Lemma 5

$$\Gamma \vdash e \in \tau$$

- If $\theta(e')$, then by (6) $l \notin \{\Downarrow, \downarrow\}$. Moreover, e diverges so it trivially belongs to the interpretation of any Div type, or by (1)

$$\Gamma \vdash e \in \tau$$

□

We define $\vdash \Gamma$ as $\vdash \emptyset$ and if $\Gamma \vdash \tau$ then $\vdash \Gamma, x : \tau$. Now, using Lemma 4 we prove substitution Lemma:

Lemma 6. *[Substitution] If* $\Gamma \vdash e_x : \tau_x$ *and* $\vdash \Gamma, x{:}\tau_x, \Gamma'$*, then*

1. *If* $\Gamma, x{:}\tau_x, \Gamma' \vdash \tau_1 \preceq \tau_2$ *then* $\Gamma, [e_x/x]\,\Gamma' \vdash [e_x/x]\,\tau_1 \preceq [e_x/x]\,\tau_2$.
2. *If* $\Gamma, x{:}\tau_x, \Gamma' \vdash e : \tau$ *then* $\Gamma, [e_x/x]\,\Gamma' \vdash [e_x/x]\,e : [e_x/x]\,\tau$.
3. *If* $\Gamma, x{:}\tau_x, \Gamma' \vdash \tau$ *then* $\Gamma, [e_x/x]\,\Gamma' \vdash [e_x/x]\,\tau$.

Proof. If $\Gamma \vdash e_x : \tau_x$ and $\Gamma, x{:}\tau_x, \Gamma' \vdash$, then

1. Assume

$$\Gamma, x{:}\tau_x, \Gamma' \vdash \tau_1 \preceq \tau_2$$

We will prove the lemma by induction on the derivation tree.

- $\preceq$-BASE Assume $\Gamma, x{:}\tau_x, \Gamma' \vdash \tau_1 \preceq \tau_2$ where $\tau_1 \equiv \{v{:}B^l \mid e_1\}$ and $\tau_2 \equiv \{v{:}B^l \mid e_2\}$. By inversion

$$\Gamma, x{:}\tau_x, \Gamma', v : B \vdash e_1 \Rightarrow e_2$$

By inversion

$$\begin{aligned}&\forall\theta, e_x', \theta', e.\theta\,[e_x'/x]\,\theta'\,[e/v] \in [\![\Gamma, x{:}\tau_x, \Gamma', v : B]\!]\\ &\quad\Rightarrow \theta\,[e_x'/x]\,\theta'\,[e/v]\,(e_1)\\ &\quad\hookrightarrow^* \texttt{true} \Rightarrow \theta\,[e_x'/x]\,\theta'\,[e/v]\,(e_2) \hookrightarrow^* \texttt{true}\end{aligned}$$

Since $\Gamma \vdash e_x : \tau_x$, so

$$\begin{aligned}&\forall\theta, \theta', e.\theta\,[e_x/x]\,\theta'\,[e/v] \in [\![\Gamma, x{:}\tau_x, \Gamma', v : B]\!]\\ &\quad\Rightarrow \theta\,[e_x/x]\,\theta'\,[e/v]\,(e_1)\\ &\quad\hookrightarrow^* \texttt{true} \Rightarrow \theta\,[e_x/x]\,\theta'\,[e/v]\,(e_2) \hookrightarrow^* \texttt{true}\end{aligned}$$

Since $\Gamma \vdash e_x : \tau_x$, so

$$\begin{aligned}&\forall\theta, \theta', e.\theta\theta'\,[e/v] \in [\![\Gamma, [e_x/x]\,\Gamma', v : B]\!]\\ &\quad\Rightarrow \theta\theta'\,[e/v]\,(e_1\,[e_x/x])\\ &\quad\hookrightarrow^* \texttt{true} \Rightarrow \theta\theta'\,[e/v]\,(e_2\,[e_x/x]) \hookrightarrow^* \texttt{true}\end{aligned}$$

So,

$$\Gamma, [e_x/x]\,\Gamma', v : B \vdash e_1\,[e_x/x] \Rightarrow e_2\,[e_x/x]$$

Or

$$\Gamma, [e_x/x]\,\Gamma', v : B \vdash t_1\,[e_x/x] \preceq t_2\,[e_x/x]$$

- $\preceq$-FUN Assume $\Gamma, x{:}\tau_x, \Gamma' \vdash \tau_1 \preceq \tau_2$, where $\tau_1 \equiv y{:}\tau_y \to \tau$ and $\tau_2 \equiv y{:}\tau_y' \to \tau'$. By inversion

$$\Gamma, x{:}\tau_x, \Gamma' \vdash \tau_y' \preceq \tau_y\ (1) \qquad \Gamma, x{:}\tau_x, \Gamma', y{:}\tau_y' \vdash \tau \preceq \tau'\ (2)$$

By IH

$$\Gamma, [e_x/x]\,\Gamma' \vdash \tau_y'\,[e_x/x] \preceq \tau_y\,[e_x/x]$$

$$\Gamma, [e_x/x]\,\Gamma', y{:}\tau_y'\,[e_x/x] \vdash \tau\,[e_x/x] \preceq \tau'\,[e_x/x]$$

By rule $\preceq$-FUN

$$\Gamma, [e_x/x]\,\Gamma' \vdash \tau_1\,[e_x/x] \preceq \tau_2\,[e_x/x]$$

2. Assume $\Gamma, x{:}\tau_x, \Gamma' \vdash e : \tau$. We will prove the lemma by induction on the derivation tree.

- T-VAR Assume $\Gamma, x{:}\tau_x', \Gamma' \vdash e : \tau$, where $e \equiv y$. By inversion

$$(y, \tau) \in \Gamma, x{:}\tau_x', \Gamma'$$

Assume

$$(y, \tau) \in \Gamma$$

By rule T-VAR

$$\Gamma, [e_x/x]\,\Gamma' \vdash e : \tau$$

Since $\vdash \Gamma$, x cannot appear in τ so $\tau\,[e_x/x] \equiv \tau$. Also, $x \neq y$, so $e\,[e_x/x] \equiv e$. So,

$$\Gamma, [e_x/x]\,\Gamma' \vdash e\,[e_x/x] : \tau\,[e_x/x]$$

Assume

$$y \equiv x$$

By lemma's assumption

$$\Gamma \vdash e_x : \tau_x$$

so

$$\Gamma, [e_x/x]\,\Gamma' \vdash e_x : \tau_x'$$

Since $x = y$, $e\,[e_x/x] \equiv e_x$. Also, since $x \notin Dom(\Gamma)$ it cannot appear in τ_x',so $\tau\,[e_x/x] \equiv \tau \equiv \tau_x'$. So,

$$\Gamma, [e_x/x]\,\Gamma' \vdash e\,[e_x/x] : \tau\,[e_x/x]$$

Otherwise, assume

$$(y, \tau) \in \Gamma'$$

So,

$$(y, [e_x/x]\,\tau) \in [e_x/x]\,\Gamma'$$

Also, $e\,[e_x/x] \equiv e \equiv y$. By which and rule T-VAR, we get

$$\Gamma, [e_x/x]\,\Gamma' \vdash e\,[e_x/x] : \tau\,[e_x/x]$$

- Case T-CON. Assume $\Gamma, x{:}\tau_x, \Gamma' \vdash e : \tau$, where $e \equiv c$ and $\tau \equiv \mathsf{Ty}(c)$. Since $e\,[e_x/x] \equiv e$ and $\tau\,[e_x/x] \equiv \tau$

$$\Gamma, [e_x/x]\,\Gamma' \vdash e\,[e_x/x] : \tau\,[e_x/x]$$

- T-SUB Assume $\Gamma, x{:}\tau_x, \Gamma' \vdash e : \tau$. By inversion

$$\Gamma, x{:}\tau_x, \Gamma' \vdash e : \tau'\ (1) \qquad \Gamma, x{:}\tau_x, \Gamma' \vdash \tau' \preceq \tau\ (2)$$

$$\Gamma, x{:}\tau_x, \Gamma' \vdash \tau\ (3)$$

  By IH, 1 and 3

$$\Gamma, [e_x/x]\,\Gamma' \vdash [e_x/x]\,e : [e_x/x]\,\tau'$$

$$\Gamma, [e_x/x]\,\Gamma' \vdash [e_x/x]\,\tau' \preceq [e_x/x]\,\tau$$

$$\Gamma, [e_x/x]\,\Gamma' \vdash [e_x/x]\,\tau$$

  By rule T-SUB

$$\Gamma, [e_x/x]\,\Gamma' \vdash e\,[e_x/x] : \tau\,[e_x/x]$$

- T-FUN Assume $\Gamma, x{:}\tau_x, \Gamma' \vdash e : \tau$, where $e \equiv \lambda y.$ and $\tau \equiv y{:}\tau'_y \to \tau'$. By inversion

$$\Gamma, x{:}\tau_x, \Gamma', y{:}\tau'_y \vdash e' : \tau'\ (1) \qquad \Gamma, x{:}\tau_x, \Gamma' \vdash \tau'_y\ (2)$$

  By IH and 3

$$\Gamma, [e_x/x]\,\Gamma', y{:}\,[e_x/x]\,\tau'_y \vdash [e_x/x]\,e' : [e_x/x]\,\tau'$$

$$\Gamma, [e_x/x]\,\Gamma' \vdash [e_x/x]\,\tau'_y$$

  By rule T-FUN

$$\Gamma, [e_x/x]\,\Gamma' \vdash [e_x/x]\,e : [e_x/x]\,\tau$$

- T-APP Assume $\Gamma, x{:}\tau_x, \Gamma' \vdash e : \tau$, where $e \equiv e_1\ e_2$ and $\tau \equiv \tau'\,[e_2/y]$. By inversion

$$\Gamma, x{:}\tau_x, \Gamma' \vdash e_1 : y{:}\tau'_y \to \tau'\ (1) \qquad \Gamma, x{:}\tau_x, \Gamma' \vdash e_2 : \tau'_y\ (2)$$

  By IH

$$\Gamma, [e_x/x]\,\Gamma' \vdash [e_x/x]\,e_1 : [e_x/x]\,y{:}\tau'_y \to \tau'$$

$$\Gamma, [e_x/x]\,\Gamma' \vdash [e_x/x]\,e_2 : [e_x/x]\,\tau'_y$$

  By rule T-APP

$$\Gamma, [e_x/x]\,\Gamma' \vdash [e_x/x]\,e : [e_x/x]\,\tau$$

- T-LET Assume $\Gamma, x{:}\tau_x, \Gamma' \vdash e : \tau$, where $e \equiv \texttt{let}\ y = e_y\ \texttt{in}\ e'$. By inversion

$$\Gamma, x{:}\tau_x, \Gamma' \vdash e_y : \tau_y\ (1) \qquad \Gamma, x{:}\tau_x, \Gamma', y{:}\tau_y \vdash e :' \tau\ (2)$$

$$\Gamma, x{:}\tau_x, \Gamma' \vdash \tau\ (3)$$

  By IH and 3

$$\Gamma, [e_x/x]\,\Gamma' \vdash e_y : \tau_y\ (4) \qquad \Gamma, [e_x/x]\,\Gamma', y{:}\tau_y \vdash e :' \tau\ (5)$$

$$\Gamma, [e_x/x]\,\Gamma' \vdash \tau\ (6)$$

  So,

$$\Gamma, [e_x/x]\,\Gamma' \vdash [e_x/x]\,e : [e_x/x]\,\tau$$

- T-CASE This case is similar to T-LET.
- T-BOT Assume $\Gamma, x{:}\tau_x, \Gamma' \vdash e : \tau$, where $e \equiv \bot$. By inversion

$$\Gamma, x{:}\tau_x, \Gamma' \vdash \tau$$

  By 3

$$[e_x/x]\,\Gamma' \vdash [e_x/x]\,\tau$$

  By rule T-BOT

$$\Gamma, [e_x/x]\,\Gamma' \vdash [e_x/x]\,e : [e_x/x]\,\tau$$

3. Assume $\Gamma, x{:}\tau_x, \Gamma' \vdash \tau$. We will prove it by induction on the derivation.

- WF-BASE Assume $\Gamma, x{:}\tau_x, \Gamma' \vdash \tau$, where $\tau \equiv \{v{:}B^l \mid e\}$. By inversion

$$\lfloor \Gamma, x{:}\tau_x, \Gamma' \rfloor, v{:}B \vdash_B e{:}\texttt{bool}$$

  So,

$$\lfloor \Gamma, [e_x/x]\,\Gamma' \rfloor, v{:}B \vdash_B e\,[e_x/x]\,{:}\texttt{bool}$$

  By rule WF-BASE

$$\Gamma, [e_x/x]\,\Gamma' \vdash \{v{:}B^l \mid e\,[e_x/x]\}$$

  Or

$$\Gamma, [e_x/x]\,\Gamma' \vdash \tau\,[e_x/x]$$

- WF-FUN Assume $\Gamma, x{:}\tau_x, \Gamma' \vdash \tau$, where $\tau \equiv y{:}\tau'_y \to \tau'$. By inversion, we get

$$\Gamma, x{:}\tau_x, \Gamma' \vdash \tau_x \qquad \Gamma, x{:}\tau_x, \Gamma', y{:}\tau'_y \vdash \tau'$$

  By IH

$$\Gamma, [e_x/x]\,\Gamma' \vdash \tau_x\,[e_x/x] \qquad \Gamma, [e_x/x]\,(\Gamma', y{:}\tau'_y) \vdash \tau'\,[e_x/x]$$

  Due to α-renaming, $x \neq y$, so

$$\Gamma, [e_x/x]\,\Gamma' \vdash \tau'_y\,[e_x/x] \qquad \Gamma, [e_x/x]\,\Gamma', y{:}\,[e_x/x]\,\tau'_y \vdash \tau'\,[e_x/x]$$

  By WF-FUN

$$\Gamma, [e_x/x]\,\Gamma' \vdash y{:}\tau'_y\,[e_x/x] \to \tau'\,[e_x/x]$$

  Or

$$\Gamma, [e_x/x]\,\Gamma' \vdash \tau\,[e_x/x]$$

□

A.3 Soundness

Figure 7 defines a $\mathsf{BotomLess}(\bullet)$ predicate on expressions:

We prove Preservation and Progress only on expressions that do not contain $\bot$:

Lemma 7 (Preservation)**.** *If* $\emptyset \vdash e : \tau$*,* $\mathsf{BotomLess}(e)$ *and* $e \hookrightarrow e'$ *then* $\emptyset \vdash e' : \tau$*.*

Proof. Helping Lemmata:

Lemma 8. *If* $\Gamma \vdash e : \tau$ *and* $\vdash \Gamma$ *then* $\Gamma \vdash \tau$*.*
Proof Sketch: By case split on the derivation $\Gamma \vdash \tau$ □

Lemma 9. *If* $e \hookrightarrow e'$ *then* $\emptyset \vdash \tau\,[e'/x] \preceq \tau\,[e/x]$
Proof Sketch: By case split on the derivation $\Gamma \vdash \tau\,[e'/x] \preceq \tau\,[e/x]$ □

Assume $\mathsf{BotomLess}(e)$ and $\emptyset \vdash e : \tau$ and $e \hookrightarrow e'$. We will prove the lemma by induction on the derivation tree.

- Cases T-VAR, T-CON, T-FUN and T-BOT trivially hold as there is no e' for which $e \hookrightarrow e'$.
- Case T-SUB. Assume $\emptyset \vdash e : \tau$. By inversion

$$\emptyset \vdash e : \tau'\ (1) \qquad \emptyset \vdash \tau' \preceq \tau\ (2) \qquad \emptyset \vdash \tau\ (3)$$

  By IH on (1)

$$\emptyset \vdash e' : \tau'$$

  By which, (2), (3) and T-SUB

$$\emptyset \vdash e' : \tau$$

- Case T-LET. Assume $\emptyset \vdash e : \tau$, where $e \equiv \texttt{let}\ x = e_x\ \texttt{in}\ e_0$. Then $e' \equiv e_0\,[e_x/x]$. By inversion

$$\emptyset \vdash e_x : \tau_x\ (1) \qquad x{:}\tau_x \vdash e_0 : \tau\ (2) \qquad \emptyset \vdash \tau\ (3)$$

  By (1), (2) and Lemma 6,

$$\emptyset \vdash e' : \tau\,[e_x/x]\ \ (4)$$

$$\mathsf{BotomLess}(c) \qquad \mathsf{BotomLess}(x) \qquad \neg\mathsf{BotomLess}(\bot)$$

$$\mathsf{BotomLess}(D\ \overline{e_i}) \Leftrightarrow \bigwedge \mathsf{BotomLess}(e_i) \qquad \mathsf{BotomLess}(\lambda x.e) \Leftrightarrow \mathsf{BotomLess}(e)$$

$$\mathsf{BotomLess}(e_1\ e_2) \Leftrightarrow \mathsf{BotomLess}(e_1) \wedge \mathsf{BotomLess}(e_2)$$

$$\mathsf{BotomLess}(\texttt{let}\ x = e_1\ \texttt{in}\ e_2) \Leftrightarrow \mathsf{BotomLess}(e_1) \wedge \mathsf{BotomLess}(e_2)$$

$$\mathsf{BotomLess}(\texttt{case}\ x = e\ \texttt{of}\ \{D_i\ \overline{x} \to e_i\}) \Leftrightarrow \mathsf{BotomLess}(e) \wedge \bigwedge \mathsf{BotomLess}(e_i)$$

Figure 7. $\mathsf{BotomLess}(e)$

By (3) x does not appear free in τ, so, $\tau\,[e_x/x] \equiv \tau$ and

$$\emptyset \vdash e' : \tau$$

- Case T-APP. Assume

$$\emptyset \vdash e : \tau\ (1)$$

where $e \equiv e_1\ e_2$, and $\tau \equiv \tau'\,[e_2/x]$
By inversion

$$\emptyset \vdash e_1 : (x{:}\tau_x \to \tau')\ (2) \qquad \emptyset \vdash e_2 : \tau_x\ (3)$$

We split cases on the structure of e.

- $e \equiv c\ v_2$. Then, $e' \equiv [\![c]\!](v_2)$. By Definition 2,

$$\emptyset \vdash e' : \tau$$

- $e \equiv c\ e_2$ where e_2 is botomless and not a value, Then, by (3) and Lemma 10, $e_2 \hookrightarrow e_2'$, and $e' \equiv e_1\ e_2'$. By IH on (3)

$$\emptyset \vdash e_2' : \tau_x$$

By which, (2) and rule T-APP we get

$$\emptyset \vdash e' : \tau'\left[e_2'/x\right]\ (4)$$

By Lemma 9

$$\emptyset \vdash \tau'\left[e_2'/x\right] \preceq \tau'\,[e_2/x]\ (5)$$

By (1) and Lemma 8, since $\vdash \emptyset$

$$\emptyset \vdash \tau'\,[e_2/x]\ (6)$$

By (4), (5), (6) and rule T-SUB

$$\emptyset \vdash e' : \tau$$

- $e \equiv \lambda x.\ e_2$. Then, $e' \equiv e_x\,[e_2/x]$.
By inversion on (2)

$$x{:}\tau_x \vdash e_x : \tau'$$

By which, (3) and Lemma 6 (since $\vdash x{:}\tau_x$)

$$\emptyset \vdash e' : \tau'$$

- $e \equiv e_1\ e_2$, where e_1 is botomless and not a value. Then, by (2) and Lemma 10, $e_1 \hookrightarrow e_1'$ and $e' \equiv e_1'\ e_2$ By IH on (2)

$$\emptyset \vdash e_1' : (x{:}\tau_x \to \tau')$$

By which, (3) and rule T-APP we get

$$\emptyset \vdash e' : \tau$$

- Case T-CASE, assume $\emptyset \vdash e : \tau$, where $e \equiv \texttt{case}\ x = e_0\ \texttt{of}\ \{D_i\ \overline{y} \to e\}$. By inversion

$$\emptyset \vdash e_T : \{v{:}T^l \mid e_T\}\ (1)$$

$$\emptyset \vdash \tau\ (2)$$

$$\forall i, 0 < i \leq \mathsf{Arity}(T).(\mathsf{Ty}(D_T^i) = x_1{:}\tau_1 \to \ldots x_n{:}\tau_n \to \{v{:}T^l \mid e_{T_i}\}\ (3)$$

$$\theta = \overline{[y_i/x_i]}\ (4) \qquad x{:}\{v{:}T \mid e_t \wedge e_{T_i}\}, \overline{y_i{:}\theta\ \tau_i} \vdash e_i : \tau\ (5))$$

We split cases on the structure of e_T.

- Assume that $e_T \equiv D_T^i\ \overline{e}$, then $e' \equiv e_i\,[e/x]\,\overline{[e_i/y_i]}$.
By (5)

$$\overline{y_i{:}\theta\ \tau_i}, x{:}\{v{:}T^l \mid e_t \wedge \theta\ e_{T_i}\} \vdash e_i : \tau$$

By inversion on (1) $\emptyset \vdash e_j : \tau_j\overline{[e/x]}$ and $\emptyset \vdash D_T^i\ \overline{e} : \{v{:}T \mid e_{T_i}\}\overline{[e/x]}$. So, $\emptyset \vdash e_j : \tau_j\overline{[y/x]\,[e/y]}$ and $\emptyset \vdash D_T^i\ \overline{e} : \{v{:}T \mid e_{T_i}\}\overline{[y/x]\,[e/y]}$. And, $\emptyset \vdash e_j \in \tau_j\overline{[y/x]\,[e/y]}$ and $\emptyset \vdash D_T^i\ \overline{e} \in \{v{:}T \mid e_{T_i}\}\overline{[y/x]\,[e/y]}$.
Finally, by Definition 2 $\emptyset \vdash D_T^i\ \overline{e} \in \{v{:}T \mid e_{T_i} \wedge e_t\}\overline{[y/x]\,[e/y]}$.
Then, by Lemma 6
$\emptyset \vdash e' : \tau\overline{[e_i/y_i]}\,\overline{[e/x]}$.
Finally, by (2), $\tau\overline{[e_i/y_i]}\,[e/x] \equiv \tau$, so

$$\emptyset \vdash e' : \tau.$$

- Otherwise, by (1) and Lemma 10 $e_0 \hookrightarrow e_0'$. So $e' \equiv \texttt{case}\ x = e_0'\ \texttt{of}\ \{D_i\ \overline{y} \to e\}$. By IH $\emptyset \vdash e_0' : \{v{:}T \mid e_T\}$, by which and (1) − (6)

$$\emptyset \vdash e' : \tau$$

□

Lemma 10 (Progress)**.** *If* $\emptyset \vdash e : \tau$, $\mathsf{BotomLess}(e)$ *and* $e \neq v$ *then there exists an* e' *such that* $\mathsf{BotomLess}(e')$ *and* $e \hookrightarrow e'$.

Proof. Assume $\emptyset \vdash e : \tau$. We will prove the Lemma by induction on the derivation tree.

- Case T-VAR cannot occur, as $\Gamma = \emptyset$
- Case T-BOT is trivial, as $\neg\mathsf{BotomLess}(e)$.
- Cases T-CON and T-FUN are trivial, as $e = v$.
- Case T-SUB. Assume $\emptyset \vdash e : \tau$. By inversion

$$\emptyset \vdash e : \tau'$$

By IH either $e \equiv v$ or there exists an botomless e' such that $e \hookrightarrow e'$.

- Case T-APP. Assume

$$\emptyset \vdash e : \tau\ (1)$$

where $e \equiv e_1\ e_2$ and $\tau \equiv \tau'\,[e_2/x]$. By inversion

$$\emptyset \vdash e_1 : (x{:}\tau_x \to \tau)\ (2) \qquad \emptyset \vdash e_2 : \tau_x\ (3)$$

We split cases on the structure of e.

- $e \equiv c\ v_2$. Then, $e' \equiv [\![c]\!](v_2)$ which is botomless by Definition of constants.
- $e \equiv c\ e_2$ where e_2 is not a value, By IH on (3) $e_2 \hookrightarrow e_2'$ and $e' \equiv e_1\ e_2'$
- $e \equiv \lambda x.\ e_2$. Then, $e' \equiv e_x\,[e_2/x]$, which does not contain bottom.

- $e \equiv e_1\ e_2$, where $e_1 \neq v$. Then, by IH on (2) $e_1 \hookrightarrow e_1'$ and $e' \equiv e_1'\ e_2$.

- Case T-LET. Assume $\emptyset \vdash e : \tau$, where $e \equiv \texttt{let}\ x = e_x\ \texttt{in}\ e_0$, then $e' \equiv e_0\,[e_x/x]$ which is botomless.
- Case T-CASE. Assume $\emptyset \vdash e : \tau$, where $e \equiv \texttt{case}\ x = e_T\ \texttt{of}\ \{D_{T_i}\ \overline{y} \to e_i\}$. By inversion,

$$\Gamma \vdash e : \{v{:}T^{e_T} \mid \}(1)$$

We split cases on the structure of e_T

- If e_T is a value, then by (1) it is of the form $e_T \equiv D_{T_i}\overline{e}$, so $e' \equiv e_i\,[e_T/x]\,\overline{[e/y]}$
- Otherwise, by IH there exists e_T' such that $e_T \hookrightarrow^* e_T'$, so $e' \equiv \texttt{case}\ x = e_T'\ \texttt{of}\ \{D_{T_i}\ \overline{y} \to e_i\}$.

□

We combine the above to prove *Soundness of* λ^U, *i.e.* Theorem 1 in the paper:

Theorem 6. *[Soundness of* λ^U *] If* $\emptyset \vdash e : \tau$ *and* $\mathsf{BotomLess}(e)$*, then*

- ***Type-Preservation:*** *If* $e \hookrightarrow^* v$ *then* $\emptyset \vdash v{:}\tau$.
- ***Crash-Freedom:*** $e \not\hookrightarrow^* \texttt{crash}$.

Proof. 1. Since $\mathsf{BotomLess}(e)$ there exists by Lemma 10 a bottomless evaluation sequence

$$e \equiv e_0 \hookrightarrow e_1 \hookrightarrow \ldots \hookrightarrow \ldots e_n \equiv v$$

The Theorem is proven by n applications of Preservation Lemma.

2. If $e \hookrightarrow^* \texttt{crash}$, then by Preservation $\emptyset \vdash \texttt{crash} : \tau$ which cannot happen, as `crash` by definition is an untyped constant. □

B. Tracking Substitutions

Then we define the notion of tracking substitutions. In Figure 8 we extend the operational semantics with a state σ, *i.e.* a mapping from variables to expressions that tracks evaluation of its expressions

First we prove that evaluation to a constant exists *iff* tracking evaluation to the same constant exists.

Lemma 11. $\forall\theta, e, c, \exists\theta'.\theta(e) \hookrightarrow^* c \Leftrightarrow \langle\theta; e\rangle \rightsquigarrow^\star \langle\theta'; c\rangle$.

Proof Sketch: We prove each direction:

- $\Rightarrow$. Given the derivation $\theta(e) \hookrightarrow^* v$, we can track the appearances of each expressions $\theta(x_i)$ and its derivatives and replace them with x_i. Thus, given the initial derivation we can transverse it (left-to-right and post-order); for every tracked appearance we use the appropriate rules that update the stack every time a tracked expressions evaluates, ie., appears in the left hand side of a rule; and remove the multiple evaluations of expressions in the stack and construct the evaluation $\langle\theta; e\rangle \rightsquigarrow^\star \langle\theta'; c\rangle$. Note that if $\theta(x_i)$ goes to a value, then $\theta(x_i) \equiv e_0 \hookrightarrow \ldots e_i \ldots e_n \equiv v$. By the way we transverse the tree, after the stack is updated to e_k and before it is updated to e_{k+1} all tracked computations for x_i are $e_j, j \leq k$.
  If $\theta(x_i)$ does not go to a value, it cannot appear in the left hand side of a rule, because evaluation would diverge, thus the stack is not updated for x_i.
  When a tracked expression reaches a value, we use the appropriate value to substitute (and untrack) the value. Since the result of the initial evaluation is a constant, then the result of the tracked computation is the same constant.
- $\Leftarrow$. Given $\langle\theta; e\rangle \rightsquigarrow^\star \langle\theta'; c\rangle$ we can construct the derivation $\theta(e) \hookrightarrow^* c$ replacing each query to the stack with the initial computation of the expression.

□

Then we define a *bottomize* function $\underline{\bullet}$ that replaces non-evaluated expressions with $\bot$:

Definition 3. *[Bottomize]*

$$\underline{\theta}(x) = \begin{cases} D\ \overline{\underline{\theta}(y)} & \text{if } \theta(x) = D\ \overline{y} \\ v & \text{if } \theta(x) = v \neq D\ \overline{y} \\ \bot & \text{otherwise} \end{cases}$$

Using the bottomize function we show that evaluation does not depend on non-evaluated expressions:

Lemma 12. *If* $\langle\theta; e\rangle \rightsquigarrow^\star \langle\theta'; c\rangle$*, then* $\underline{\theta'}\ e \hookrightarrow^* c$.

Proof Sketch: Since $\langle\theta; e\rangle \rightsquigarrow^\star \langle\theta'; c\rangle(1)$, then $\langle\theta'; e\rangle \rightsquigarrow^\star \langle\theta'; c\rangle(2)$: From the evaluation tree (1) we can construct the evaluation tree (2). The trees differ on store related rules.

Say that in (1) the store in x is updated, for an arbitrary x: $\langle(x, e_x)\theta_x; x\rangle \rightsquigarrow \langle(x, e_x')\theta_x; x\rangle$ Since (1) is finite, it should be that $\langle\theta_x; e_x\rangle \rightsquigarrow^\star \langle\theta_x; v\rangle(3)$. Call $v_x = D\ \overline{y}$ if $v = D\ \overline{e}$, v otherwise. Then in (1) there should be a "subtree" with (3) after which the value of x cannot change in the store. Or $\theta'(x) = v_x$. We construct (2) by removing the "subtree" with (3). After that all rules that relate store with x will be the same on (1) and (2).

If x is not updated in (1) then x does not appear in the left hand side of a rule; thus $\theta'(x) = \theta(x)$.

We construct $\theta''(x) = \begin{cases} v & \text{if } \theta'(x) = v \\ \bot & \text{otherwise} \end{cases}$

Then $\langle\theta''; e\rangle \rightsquigarrow^\star \langle\theta''; c\rangle$. If $\theta'(x)$ is not a value, then it does not appear in the left hand side of any rule in (2), thus evaluation of e cannot depend on x.

Then by Lemma 11, $\theta''(e) \hookrightarrow^* c$. But $\underline{\theta'}e = \theta''(e)$, so $\underline{\theta'}(e) \hookrightarrow^* c$. □

Also, replacing $\bot$ with any expression yields the same evaluation:

Lemma 13. *If* $\underline{\theta}(e) \hookrightarrow^* c$*, then* $\theta(e) \hookrightarrow^* c$.

Proof. Since $\underline{\theta}(e) \hookrightarrow^* c(1)$, then $\langle\underline{\theta}; e\rangle \rightsquigarrow^\star \langle\theta'; c\rangle(2)$. $\bot$ expressions in $\underline{\theta}$ are not evaluated, otherwise (2) would get stuck. Thus they can be instantiated with any expression. θ provides such an instantiation, thus $\langle\theta; e\rangle \rightsquigarrow^\star \langle\theta''; c\rangle(3)$. By Lemma 11, $\theta(e) \hookrightarrow^* c$. □

Finally, we define lifting substitutions

Definition 4. *[Lifting Substitutions]* $\theta \hookrightarrow^*_\bot \theta^\bot \doteq \exists e, e', \theta'\,\langle\theta; e\rangle \rightsquigarrow^\star \langle\theta'; e'\rangle \wedge \theta^\bot = \underline{\theta'}$

and prove the Lifting Lemma

Lemma 14. *[Lifting]* $\theta(e) \hookrightarrow^* c$ *iff* $\exists\theta \hookrightarrow^*_\bot \theta^\bot$ *s.t.* $\theta^\bot(e) \hookrightarrow^* c$.

Proof Sketch: The $\Rightarrow$ direction follows immediately from Lemmata 11 and 12. The $\Leftarrow$ direction follows immediately from Lemmata 11 and 13. □

C. Constants

We can prove that all the above constants belong to the interpretations of their types.

Theorem 7. *[Constants]* $c \in [\![\mathsf{Ty}(c)]\!]$.

The Theorem trivially holds for more of the constants. For example,

$$=\in [\![x{:}b^{\Downarrow} \to y{:}b^{\Downarrow} \to \{v{:}\texttt{bool}^{\Downarrow} \mid v \Leftrightarrow x = y\}]\!]$$

as $\forall e_1, e_2, e_1 \hookrightarrow^* d_1, e_2 \hookrightarrow^* d_2 \Rightarrow (e_1 = e_2 \Leftrightarrow e_1 = e_2) \hookrightarrow^* \texttt{true} \wedge \exists d.(e_1 = e_2) \hookrightarrow^* d$

Here we prove that for any type τ, $\texttt{fix}_\tau$ and $\texttt{tfix}_\tau$ satisfy Theorem 7.

$$\langle \sigma; e \rangle \rightsquigarrow \langle \sigma; e \rangle$$

$$\begin{array}{ll}
\langle \sigma; e_1\ e_2 \rangle \rightsquigarrow \langle \sigma'; e_1'\ e_2 \rangle & \text{if } \langle \sigma; e_1 \rangle \rightsquigarrow \langle \sigma'; e_1' \rangle \\
\langle \sigma; c\ e \rangle \rightsquigarrow \langle \sigma'; c\ e' \rangle & \text{if } \langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle \\
\langle \sigma; \texttt{case}\ x = e\ \texttt{of}\ \{D_i\ \overline{y_i} \to e_i\} \rangle \rightsquigarrow \langle \sigma'; \texttt{case}\ x = e'\ \texttt{of}\ \{D_i\ \overline{y_i} \to e_i\} \rangle & \text{if } \langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle \\
\langle \sigma; D\ \overline{e_i}\ e\ \overline{e_j} \rangle \rightsquigarrow \langle \sigma'; D\ \overline{e_i}\ e'\ \overline{e_j} \rangle & \text{if } \langle \sigma; e \rangle \rightsquigarrow \langle \sigma'; e' \rangle \\
\langle \sigma; c\ v \rangle \rightsquigarrow \langle \sigma; \delta(c, v) \rangle & \\
\langle \sigma; \lambda x.e\ e_x \rangle \rightsquigarrow \langle \sigma; e\,[e_x/x] \rangle & \\
\langle \sigma; \texttt{let}\ x = e_x\ \texttt{in}\ e \rangle \rightsquigarrow \langle \sigma; e\,[e_x/x] \rangle & \\
\langle \sigma; \texttt{case}\ x = D_j\ \overline{e}\ \texttt{of}\ \{D_i\ \overline{y_i} \to e_i\} \rangle \rightsquigarrow \langle \sigma; e_j\,[D_j\ \overline{e}/x]\,[\overline{e}/\overline{y_j}] \rangle & \\
\langle (x, e_x)\sigma; x \rangle \rightsquigarrow \langle (x, e_x')\sigma'; x \rangle & \text{if } \langle \sigma; e_x \rangle \rightsquigarrow \langle \sigma'; e_x' \rangle \\
\langle (y, e_y)\sigma; x \rangle \rightsquigarrow \langle (y, e_y)\sigma'; e_x \rangle & \text{if } \langle \sigma; x \rangle \rightsquigarrow \langle \sigma'; e_x \rangle \\
\langle (x, v)\sigma; x \rangle \rightsquigarrow \langle (x, v)\sigma; v \rangle & \text{if } v \neq D\ \overline{e} \\
\langle (x, D\ \overline{y})\sigma; x \rangle \rightsquigarrow \langle (x, D\ \overline{y})\sigma; D\ \overline{y} \rangle & \\
\langle (x, D\ \overline{e})\sigma; x \rangle \rightsquigarrow \langle (x, D\ \overline{y})(y_i, e_i)\sigma; D\ \overline{y} \rangle & \text{if fresh } \overline{y_i}
\end{array}$$

Figure 8. Tracking Substitutions

Given the families of constants:

$$\delta(\texttt{tfix}_\tau, f) \doteq \lambda n.\lambda f.\texttt{tfix}_\tau^n$$
$$\delta(\texttt{tfix}_\tau^n, m) \doteq \lambda f.f\ m\ (\texttt{tfix}_\tau^m\ f)$$

and their types

$$\mathsf{Ty}(\texttt{tfix}_\tau) \doteq (n{:}\texttt{nat}^{\Downarrow} \to \tau_n \to \tau\,[n/x]) \to m{:}\texttt{nat}^{\Downarrow} \to \tau\,[m/x]$$
$$\mathsf{Ty}(\texttt{tfix}_\tau^n) \doteq (n{:}\texttt{nat}^{\Downarrow} \to \tau_n \to \tau\,[n/x]) \to \tau_n$$

we prove that the constants belong to the meanings of their types:

Theorem 8. *[Terminating Fixpoint]*

1. $\forall n.\texttt{tfix}_\tau^n \in \mathsf{Ty}(\texttt{tfix}_\tau^n)$
2. $\texttt{tfix}_\tau \in \mathsf{Ty}(\texttt{tfix}_\tau)$
3. $\texttt{fix}_\tau \in \mathsf{Ty}(\texttt{fix}_\tau)$*, if the result of* τ *is a Div type.*

Proof.
- 1. We prove that for all appropriate f and $m \in [\![\{v{:}\texttt{nat}^{\Downarrow} \mid v < n\}]\!]$, $e \equiv \texttt{tfix}_\tau^n\ f\ m \in [\![\tau\,[m/x]]\!]$ by induction on n.
  For $n = 0$, it is trivial, as there is no m such that $m \in [\![\{v{:}\texttt{nat}^{\Downarrow} \mid v < 0\}]\!]$.
  For the inductive step, e reduces to
  $$\texttt{tfix}_\tau^n\ f\ m \hookrightarrow \texttt{tfix}_\tau^{n,f}\ m \hookrightarrow f\ m\ (\texttt{tfix}_\tau^m\ f)$$
  By IH, since $m < n$, $\texttt{tfix}_\tau^m \in \mathsf{Ty}(\texttt{tfix}_\tau^m)$, so f receives the appropriate arguments, and returns the appropriate result that proves the theorem.
- 2. We prove that for all appropriate f and $m \in [\![\texttt{nat}^{\Downarrow}]\!]$, $\texttt{tfix}_\tau\ f\ m \in [\![\tau\,[m/x]]\!]$. Since $m \in [\![\{v{:}\texttt{nat}^{\Downarrow} \mid v < m+1\}]\!]$
  $$\texttt{tfix}_\tau^{m+1}\ f\ m \in [\![\tau\,[m/x]]\!]$$
  But operationally, $\texttt{tfix}_\tau^{m+1}\ f\ m$ and $\texttt{tfix}_\tau\ f\ m$ behave equivalently, which proves the theorem.
- 3. The prove for $\texttt{fix}_\tau \in \mathsf{Ty}(\texttt{fix}_\tau)$. is similar. The only difference is that for the base case $\texttt{fix}_\tau^0$ should be defined to belong into the interpretation of any type. Thus, it is defined as a diverging expression and the type of $\texttt{fix}_\tau$ is constrainted to τ's with potentially diverging result. With refinement types we prove that the basic $\texttt{tfix}_\tau^0$ operator cannot be called, so we omit the definition of this basic case.

□

D. Algorithmic Typing: λ^D

Soundness of λ^D trivially reduces to soundness of implication checking. Here we give the detailed proof of the Approximation Theorem:

Theorem 9. *[Approximation] If* $\Gamma \vdash_D p_1 \Rightarrow p_2$ *then* $\Gamma \vdash p_1 \Rightarrow p_2$.

Proof. To prove the above, let $VC \doteq (\!|\Gamma|\!) \Rightarrow (\!|p_1|\!) \Rightarrow (\!|p_2|\!)$. First, note that if VC is u-valid then it is valid as the addition of axioms preserves validity. Next, we prove that if the VC is valid then $\Gamma \vdash p_1 \Rightarrow p_2$. We fix a θ for which $\theta \in [\![\Gamma]\!]$ and $\theta(q_1) \hookrightarrow^* \texttt{true}$ It suffices to prove that $\theta(q_2) \hookrightarrow^* \texttt{true}$.

For all $(x_i, \{x_i{:}B^{\downarrow} \mid p_i\}) \in \Gamma$ there exists $(x_i, e_i) \in \theta$ and

$$\begin{aligned}
e_i \in [\![\{x_i{:}B^{\downarrow} \mid p_i\}]\!] &\Leftrightarrow e_i \in [\![\{v{:}B \mid p_i\}]\!] \wedge \exists v.e_i \hookrightarrow^* v_i \\
&\Leftrightarrow \exists v.e_i \hookrightarrow^* v \Rightarrow \theta(p_i\,[v_i/x_i]) \hookrightarrow^* \texttt{true} \wedge \exists v.e_i \hookrightarrow^* v_i \\
&\Leftrightarrow \theta(p_i) \hookrightarrow^* \texttt{true}
\end{aligned}$$

Thus we have that $\theta\ (\bigwedge p_i \wedge q_1) \hookrightarrow^* \texttt{true}$. By Lemma 11 $\langle \theta; (\bigwedge p_i \wedge q_1) \rangle \rightsquigarrow^\star \langle \theta'; \texttt{true} \rangle$. Let $\rho = \underline{\theta'}$; thus, $\theta \hookrightarrow_\perp^* \rho$. By Lemma 14 $\rho(\bigwedge p_i \wedge q_1) \hookrightarrow^* \texttt{true}$. Moreover, by the construction of ρ, $\emptyset \vdash \rho(\bigwedge p_i \wedge q_1) : \texttt{bool}$. Thus, by Equivalence Theorem 10 $\forall \sigma \in (\!|\rho|\!).\sigma \models \bigwedge p_i \wedge q_1$. By which and validity of VC $\forall \sigma \in (\!|\rho|\!).\sigma \models q_2$. Using the other direction of Equivalence Theorem 10 $\rho\ q_2 \hookrightarrow^* \texttt{true}$. Finally, using the other direction of Lemma 14 $\theta(q_2) \hookrightarrow \texttt{true}$. □

To conclude the proof we prove Equivalence Theorem. Let $\theta^\perp$be a substitution from variables to *lifted values*. We define the embedding of the substitution $(\!|\theta^\perp|\!)$ that maps $\perp$ to arbitrary elements of the logical domain:

Definition 5.

$$(\!|\rho|\!) = \{(x_1, v_1^\perp), \ldots, (x_n, v_n^\perp) \mid v_i^\perp \in (\!|\rho(x_i)|\!)\}$$

$$(\!|\perp|\!) = \mathcal{D} \qquad (\!|D\ \overline{v}|\!) = \{D\ \overline{v} \mid v_i \in (\!|v|\!)\}$$
$$(\!|n|\!) = \{n\} \qquad (\!|v|\!) = \{c_v\}, \textit{otherwise}$$

Then we prove that given a lifted substitution a predicate goes to `true` if and only if for any embedding the predicate holds.

Theorem 10. *[Equivalence] If* $\emptyset \vdash \theta^\perp(p) : \texttt{bool}$*, then*

- $\theta^\perp(p) \hookrightarrow^* \texttt{true}$ *iff* $\forall \sigma \in (\!|\theta^\perp|\!).\sigma \models p$.
- $\theta^\perp(p) \hookrightarrow^* \texttt{false}$ *iff* $\forall \sigma \in (\!|\theta^\perp|\!).\sigma \not\models p$.

Proof. To begin with we define a comparison between lifted values and elements of the logical domain:

Definition 6.

$$\bot \sqsubset d \qquad d \sqsubseteq d \qquad v \sqsubseteq c_v$$

and a function $\mathcal{I}_\sigma(t)$ that given a model σ and an (open) logical term t returns an element in the logic:

Definition 7. *[Interpretation]*

$$\mathcal{I}_\sigma :: t \to d$$

$$\mathcal{I}_\sigma(n) = n \qquad \mathcal{I}_\sigma(f\ \bar{t}) = f_D\ (\overline{\mathcal{I}_\sigma t})$$
$$\mathcal{I}_\sigma(D\ \bar{t}) = D\ \overline{\mathcal{I}_\sigma t}$$
$$\mathcal{I}_\sigma(x) = \sigma(x) \qquad \mathcal{I}_\sigma(t_1 \oplus t_2) = \mathcal{I}_\sigma t_2 \oplus_D \mathcal{I}_\sigma t_2$$

We relate the evaluation of logical terms with their interpretation into the logic:

Lemma 15. *If $\Gamma \vdash \theta^\bot(t) : \tau$, then $\theta^\bot(t) \hookrightarrow^* v^\bot \Leftrightarrow \forall \sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t) \sqsupseteq v^\bot$*

Proof. By induction on the structure of t.

- $t \equiv n$: $\rho\ n \hookrightarrow^* n$ and $\forall \sigma \in (\!|\rho|\!)\mathcal{I}_\sigma(n) = n$
- $t \equiv x$: $\rho(x) \hookrightarrow^* \rho(x)$ and $\forall \sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(x) = \sigma(x) \sqsupseteq \rho(x)$
- $t \equiv f\ \bar{t}$:

$$\rho\ (f\ \bar{t}) \hookrightarrow^* v^\bot \Leftrightarrow f\ \overline{\rho\ t} \hookrightarrow^* v^\bot \Leftrightarrow$$
$$\exists v_i^\bot.\rho(t_i) \hookrightarrow^* v_i^\bot \text{ and } f(\overline{v^\bot}) \hookrightarrow^* v^\bot \Leftrightarrow$$
$$\exists v_i^\bot \forall \sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_i) \sqsupseteq v_i^\bot \text{ and } \forall d_i \sqsupseteq v_i^\bot.f_D(\bar{d}) \sqsupseteq v^\bot \overset{(*)}{\Leftrightarrow}$$
$$\forall \sigma \in (\!|\rho|\!).\exists d_i \mathcal{I}_\sigma(t_i) = d_i \text{ and } f_D(\bar{d}) \sqsupseteq v^\bot \Leftrightarrow$$
$$\forall \sigma \in (\!|\rho|\!).f_D(\overline{\mathcal{I}_\sigma(t_i)}) \sqsupseteq v^\bot \Leftrightarrow$$
$$\forall \sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(f\ \bar{t}) \sqsupseteq v^\bot$$

  $(*)$ We can show that for each f_D and $v^\bot$

$$\exists v_i^\bot \forall d_i.d_i \sqsupseteq v_i^\bot \Leftrightarrow f_D(\bar{d}) \sqsupseteq v^\bot$$

  ie, $v_i^\bot$ contains the least information required by f_D to produce a result less than $v^\bot$. Now, say

$$\exists \sigma \in (\!|\rho|\!) \forall d_i.\mathcal{I}_\sigma(t_i) = d_i \not\sqsupseteq v_i^\bot$$

  Then, by definition of $v_i^\bot$, $f_D(\bar{d}) \not\sqsupseteq v^\bot$, which is a contradiction.
- $t \equiv D\ \bar{t}$:

$$\rho\ (D\ \bar{t}) \hookrightarrow^* D\ \overline{v^\bot} \Leftrightarrow \rho\ t_i \hookrightarrow^* v_i^\bot \Leftrightarrow$$
$$\forall \sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_i) \sqsupseteq v_i^\bot \Leftrightarrow \forall \sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(D\ \bar{t}) \sqsupseteq D\ \overline{v^\bot}$$

- $t \equiv t_1 \oplus t_2$

$$\rho\ (t_1 \oplus t_2) \hookrightarrow^* d \Leftrightarrow (\rho\ t_1) \oplus (\rho\ t_2) \hookrightarrow^* d \Leftrightarrow$$
$$\exists d_1.\rho\ t_1 \hookrightarrow^* d_1 \text{ and } \oplus_{d_1}\ (\rho\ t_2) \hookrightarrow^* d \Leftrightarrow$$
$$\exists d_1, d_2.\rho\ t_1 \hookrightarrow^* d_1 \text{ and } \rho\ t_2 \hookrightarrow^* d_2 \text{ and } d_1 \oplus_D d_2 = d \Leftrightarrow$$
$$\exists d_1, d_2.\forall \sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_1) = d_1 \text{ and } \forall \sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_2) = d_2 \text{ and } d_1 \oplus_D d_2 = d \overset{(*)(*)}{\Leftrightarrow}$$
$$\forall \sigma \in (\!|\rho|\!).\exists d_1, d_2.\mathcal{I}_\sigma(t_1) = d_1 \text{ and } \mathcal{I}_\sigma(t_2) = d_2 \text{ and } d_1 \oplus_D d_2 = d \Leftrightarrow$$
$$\forall \sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_1 \oplus t_2) = d$$

  $(*)$ For $i = 1, 2$, fix two instantiations $\sigma_1, \sigma_2 \in (\!|\rho|\!)$. Assume that $d_{i_{\sigma_1}} \neq d_{i_{\sigma_2}}$. Then $\neg \forall \sigma \in (\!|\rho|\!)\mathcal{I}_\sigma(t_i) = d \Rightarrow \rho\ t_i \not\hookrightarrow^* d \Rightarrow \neg\Gamma \vdash t_i : b^\Downarrow \Rightarrow \neg\Gamma \vdash p : \texttt{bool}$.

□

We use the above Lemma to prove the Theorem by induction on the structure of p.

- $p \equiv \texttt{true}$:
  - $\rho\ \texttt{true} \hookrightarrow \texttt{true}$ and $\forall \forall \sigma \in (\!|\rho|\!).\sigma \models \texttt{true}$
  - $\rho\ \texttt{true} \not\hookrightarrow \texttt{false}$ and $\exists \forall \sigma \in (\!|\rho|\!).\sigma \models \texttt{true}$
- $p \equiv \texttt{false}$:
  - $\rho\ \texttt{false} \not\hookrightarrow \texttt{true}$ and $\exists \sigma \in (\!|\rho|\!).\sigma \not\models \texttt{false}$
  - $\rho\ \texttt{false} \hookrightarrow \texttt{false}$ and $\forall \sigma \in (\!|\rho|\!).\sigma \not\models \texttt{false}$
- $p \equiv \neg q$:
  - $\rho\ (\neg q) \hookrightarrow^* \texttt{true} \Leftrightarrow \neg(\rho\ q) \hookrightarrow^* \texttt{true} \Leftrightarrow \rho\ q \hookrightarrow^* \texttt{false} \Leftrightarrow \forall \sigma \in (\!|\rho|\!).\sigma \not\models q \Leftrightarrow \forall \sigma \in (\!|\rho|\!).\sigma \models \neg q \Leftrightarrow \forall \sigma \in (\!|\rho|\!).\sigma \models p$
  - $\rho\ (\neg q) \hookrightarrow^* \texttt{false} \Leftrightarrow \neg(\rho\ q) \hookrightarrow^* \texttt{false} \Leftrightarrow \rho\ q \hookrightarrow^* \texttt{true} \Leftrightarrow \forall \sigma \in (\!|\rho|\!).\sigma \models q \Leftrightarrow \forall \sigma \in (\!|\rho|\!).\sigma \not\models \neg q \Leftrightarrow \forall \sigma \in (\!|\rho|\!).\sigma \not\models p$
- $p \equiv p_1 \wedge p_2$:
  - $\rho\ (p_1 \wedge p_2) \hookrightarrow^* \texttt{true} \Leftrightarrow (\rho\ p_1) \wedge (\rho\ p_2) \hookrightarrow^* \texttt{true} \Leftrightarrow \rho\ p_1 \hookrightarrow^* \texttt{true}$ and $\rho\ p_2 \hookrightarrow^* \texttt{true} \Leftrightarrow \forall \sigma \in (\!|\rho|\!).\sigma \models p_1$ and $\forall \sigma \in (\!|\rho|\!).\sigma \models p_2 \Leftrightarrow \forall \sigma \in (\!|\rho|\!).(\!|\rho|\!) \models p_1 \wedge p_2 \Leftrightarrow \forall \sigma \in (\!|\rho|\!).(\!|\rho|\!) \models p$
  - $\rho\ (p_1 \wedge p_2) \hookrightarrow^* \texttt{false} \Leftrightarrow (\rho\ p_1) \wedge (\rho\ p_2) \hookrightarrow^* \texttt{false} \Leftrightarrow \left\{ \begin{array}{ccc} \rho\ p_1 \hookrightarrow^* \texttt{false} & & \forall \sigma \in (\!|\rho|\!).\sigma \not\models p_1 \\ \text{OR} & \Leftrightarrow & \text{OR} \\ \rho\ p_2 \hookrightarrow^* \texttt{false} & & \forall \sigma \in (\!|\rho|\!).\sigma \not\models p_2 \end{array} \right\} \Leftrightarrow \forall \sigma \in (\!|\rho|\!).\sigma \not\models p_1 \wedge p_2 \Leftrightarrow \forall \sigma \in (\!|\rho|\!).\sigma \not\models p$
- $p \equiv t_1 = t_2$:

$$\begin{array}{lll} & \rho\ (t_1 = t_2) \hookrightarrow^* \texttt{true} & \\ \Leftrightarrow & (\rho\ t_1) = (\rho\ t_2) \hookrightarrow^* \texttt{true} & \\ \Leftrightarrow & \exists d_1, d_2.\rho\ t_1 \hookrightarrow^* d_1 & \text{and } =_{d_1}\ (\rho\ t_2) \hookrightarrow^* \texttt{true} \\ \Leftrightarrow & \exists d_1, d_2.\rho\ t_1 \hookrightarrow^* d_1 & \text{and } \rho\ t_2 \hookrightarrow^* d_2 \text{ and } d_1 =_D d_2 \\ \Leftrightarrow & \exists d_1, d_2 \forall \sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_1) = d_1 & \text{and } \forall \sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_2) = d_2 \text{ and } d_1 =_D d_2 \\ \overset{(*)}{\Leftrightarrow} & \forall \sigma \in (\!|\rho|\!) \exists d_1, d_2.\mathcal{I}_\sigma(t_1) = d_1 & \text{and } \mathcal{I}_\sigma(t_2) = d_2 \text{ and } d_1 =_D d_2 \\ \Leftrightarrow & \forall \sigma \in (\!|\rho|\!).\sigma \models t_1 = t_2 & \end{array}$$

$$\begin{array}{lll} & \rho\ (t_1 = t_2) \hookrightarrow^* \texttt{false} & \\ \Leftrightarrow & (\rho\ t_1) = (\rho\ t_2) \hookrightarrow^* \texttt{false} & \\ \Leftrightarrow & \exists d_1, d_2.\rho\ t_1 \hookrightarrow^* d_1 & \text{and } =_{d_1}\ (\rho\ t_2) \hookrightarrow^* \texttt{false} \\ \Leftrightarrow & \exists d_1, d_2.\rho\ t_1 \hookrightarrow^* d_1 & \text{and } \rho\ t_2 \hookrightarrow^* d_2 \text{ and } d_1 \neq_D d_2 \\ \Leftrightarrow & \exists d_1, d_2 \forall \sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_1) = d_1 & \text{and } \forall \sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_2) = d_2 \text{ and } d_1 \neq_D d_2 \\ \overset{(*)(*)}{\Leftrightarrow} & \forall \sigma \in (\!|\rho|\!) \exists d_1, d_2.\mathcal{I}_\sigma(t_1) = d_1 & \text{and } \mathcal{I}_\sigma(t_2) = d_2 \text{ and } d_1 \neq_D d_2 \\ \Leftrightarrow & \forall \sigma \in (\!|\rho|\!).\sigma \not\models t_1 = t_2 & \end{array}$$

  $(*)$ For $i = 1, 2$, fix two instantiations $\sigma_1, \sigma_2 \in (\!|\rho|\!)$. Assume that $d_{i_{\sigma_1}} \neq d_{i_{\sigma_2}}$. Then $\neg \forall \sigma \in (\!|\rho|\!)\mathcal{I}_\sigma(t_i) = d \Rightarrow \rho\ t_i \not\hookrightarrow^* d \Rightarrow \neg\Gamma \vdash t_i : b^\Downarrow \Rightarrow \neg\Gamma \vdash p : \texttt{bool}$
- $p \equiv t_1 < t_2$:

$$
\begin{array}{lll}
 & \rho\,(t_1 < t_2) \hookrightarrow^* \texttt{true} & \\
\Leftrightarrow & (\rho\,t_1) < (\rho\,t_2) \hookrightarrow^* \texttt{true} & \\
\Leftrightarrow & \exists d_1.\rho\,t_1 \hookrightarrow^* d_1 & \text{and} \quad <_{d_1} (\rho\,t_2) \hookrightarrow^* \texttt{true} \\
\Leftrightarrow & \exists d_1, d_2.\rho\,t_1 \hookrightarrow^* d_1 & \text{and} \quad \rho\,t_2 \hookrightarrow^* d_2 \\
 & & \text{and} \quad d_1 <_D d_2 \\
\Leftrightarrow & \exists d_1, d_2.\forall(\!|\rho|\!).\mathcal{I}_\sigma(t_1) = d_1 & \text{and} \quad \forall\sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_2) = d_2 \\
 & & \text{and} \quad d_1 <_D d_2 \\
\Leftrightarrow & \exists d_1, d_2.\forall\sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_1) = d_1 & \text{and} \quad \mathcal{I}_\sigma(t_2) = d_2 \\
 & & \text{and} \quad d_1 <_D d_2 \\
\overset{(*)}{\Leftrightarrow} & \forall\sigma \in (\!|\rho|\!)\exists d_1, d_2.\mathcal{I}_\sigma(t_1) = d_1 & \text{and} \quad \mathcal{I}_\sigma(t_2) = d_2 \\
 & & \text{and} \quad d_1 <_D d_2 \\
\Leftrightarrow & \forall\sigma \in (\!|\rho|\!).\sigma \models t_1 < t_2 & \\
 & \rho\,(t_1 < t_2) \hookrightarrow^* \texttt{false} & \\
\Leftrightarrow & (\rho\,t_1) < (\rho\,t_2) \hookrightarrow^* \texttt{false} & \\
\Leftrightarrow & \exists d_1.\rho\,t_1 \hookrightarrow^* d_1 & \text{and} \quad <_{d_1} (\rho\,t_2) \hookrightarrow^* \texttt{false} \\
\Leftrightarrow & \exists d_1, d_2.\rho\,t_1 \hookrightarrow^* d_1 & \text{and} \quad \rho\,t_2 \hookrightarrow^* d_2 \\
 & & \text{and} \quad d_1 \not<_D d_2 \\
\Leftrightarrow & \exists d_1, d_2\forall\sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_1) = d_1 & \text{and} \quad \forall\sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_2) = d_2 \\
 & & \text{and} \quad d_1 \not<_D d_2 \\
\Leftrightarrow & \exists d_1, d_2\forall\sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t_1) = d_1 & \text{and} \quad \mathcal{I}_\sigma(t_2) = d_2 \\
 & & \text{and} \quad d_1 \not<_D d_2 \\
\overset{(*)}{\Leftrightarrow} & \forall\sigma \in (\!|\rho|\!)\exists d_1, d_2.\mathcal{I}_\sigma(t_1) = d_1 & \text{and} \quad \mathcal{I}_\sigma(t_2) = d_2 \\
 & & \text{and} \quad d_1 \not<_D d_2 \\
\Leftrightarrow & \forall\sigma \in (\!|\rho|\!).\sigma \not\models t_1 < t_2 &
\end{array}
$$

$(*)$ For $i = 1, 2$, fix two instantiations $\sigma_1, \sigma_2 \in (\!|\rho|\!)$. Assume that $d_{i_{\sigma_1}} \neq d_{i_{\sigma_2}}$. Then $\neg\forall\sigma \in (\!|\rho|\!)\mathcal{I}_\sigma(t_i) = d \Rightarrow \rho\,t_i \not\hookrightarrow^* d \Rightarrow \neg\Gamma \vdash t_i : b^{\Downarrow} \Rightarrow \neg\Gamma \vdash p : \texttt{bool}$

- $p \equiv t$:
  - $\rho\,t \hookrightarrow^* \texttt{true} \Leftrightarrow \forall\sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t) = \texttt{true} \Leftrightarrow \forall\sigma \in (\!|\rho|\!).\sigma \models t$
  - $\rho\,t \hookrightarrow^* \texttt{false} \Leftrightarrow \forall\sigma \in (\!|\rho|\!).\mathcal{I}_\sigma(t) = \texttt{false} \Leftrightarrow \forall\sigma \in (\!|\rho|\!).\sigma \not\models t$

□

E. Implementation: LIQUIDHASKELL

Here we give some more examples on how we can use LIQUIDHASKELL. We start by proving termination on mutual recursive functions, using lexicographical ordering. Then we describe how we proved functional correctness on two commonly used functions, namely `ByteString` and `Text`.

E.1 Proving Termination

Next, consider the Ackermann function.

```
ack m n
  | m == 0     = n + 1
  | n == 0     = ack (m-1) 1
  | otherwise = ack (m-1) (ack m (n-1))
```

There exists no integer termination metric that decreases at each recursive call. However `ack` terminates because at each call *either* `m` decreases *or* `m` remains the same and `n` decreases. In other words, the pair `(m,n)` strictly decreases according to *lexicographic* ordering. To capture this requirement we extend termination metric from an integer to a list of integers and at each recursive call we check that this list is lexicographically decreasing. In the case of `ack` this list will simply be the parameters `m` and `n`:

```
ack :: m:Nat -> n:Nat -> Nat / [m,n]
```

Thus, LIQUIDHASKELL uses lexicographic ordering on a list of natural numbers to prove termination. Termination metrics could be generalized to any *well-found* metric.

Mutual Recursion Equipped with termination metrics LIQUIDHASKELL instantiates a powerful termination checker that like [40] proves termination even for mutual recursive functions. Consider the mutual recursive functions `isEven` and `isOdd`

```
{-@ isEven :: n:Nat -> Bool / [n, 0] @-}
{-@ isOdd  :: n:Nat -> Bool / [n, 1] @-}

isEven 0 = True
isEven n = isOdd $ n-1

isOdd n  = not $ isEven n
```

Each call terminates as either `isEven` calls `isOdd` with a decreasing argument, or the argument remains the same, and `isOdd` calls `isEven` that should then decrease the argument. We capture this reasoning using two lexicographic pairs: each function has its own metric, and when `isEven` calls `isOdd` the metric of the caller $(n, 1)$ should be greater that callee's metric $(n-1, 1)$. Similarly, at `isEven`'s call-site LIQUIDHASKELL verifies that $(n, 1) > (n, 0)$. For example, the call `isEven m` will fire the decreasing metric sequence $(m, 1) > (m, 0) > (m-1, 1) > (m-1, 0) > (m-2, 1) > \ldots$ that ultimate terminates for *any* natural number m.

E.2 Bytestring

The terms "Haskell" and "pointer arithmetic" rarely occur in the same sentence. Thus, from a verification point of view, the single most important aspect of the `Bytestring` library [28], our first case study, is its pervasive intermingling of high level abstractions like higher-order loops, folds, and fusion, with low-level pointer manipulations in order to achieve high-performance. `Bytestring` is an appealing target for evaluating LIQUIDHASKELL, as refinement types are an ideal way to statically ensure the correctness of the delicate pointer manipulations, errors in which lie below the scope of dynamic protection.

The library spans 8 files (modules) totaling about 3600 lines. We used LIQUIDHASKELL to verify the library by giving precise types describing the sizes of internal pointers and bytestrings. These types are used in a modular fashion to verify the implementation of functional correctness properties of higher-level API functions which are built using lower-level internal operations. Next, we show the key invariants and how LIQUIDHASKELL reasons precisely about pointer arithmetic and higher-order codes.

Key Invariants A (strict) `ByteString` is a triple of a `pay`load pointer, an `off`set into the memory buffer referred to by the pointer (at which the string actually "begins") and a `len`gth corresponding to the number of bytes in the string, which is the size of the buffer *after* the `off`set, that corresponds to the string. We define a measure for the *size* of a `ForeignPtr`'s buffer, and use it to define the key invariants as a refined datatype

```
measure fplen  :: ForeignPtr a -> Int
data ByteString = PS
 { pay :: ForeignPtr Word8
 , off :: {v:Nat | v       <= (fplen pay)}
 , len :: {v:Nat | off + v <= (fplen pay)} }
```

The definition states that the offset is a `Nat` no bigger than the size of the `payload`'s buffer, and that the sum of the `off`set and non-negative `len`gth is no more than the size of the payload buffer. Finally, we encode a `ByteString`'s size as a measure.

```
measure bLen   :: ByteString -> Int
bLen (PS p o l) = l
```

Specifications We define a type alias for a `ByteString` whose length is the same as that of another, and use the alias to type the API function `copy`, which clones `ByteString`s.

```
type ByteStringEq B
  = {v:ByteString | (bLen v) = (bLen B)}
```

```
copy :: b:ByteString -> ByteStringEq b
copy (PS fp off len)
  = unsafeCreate len $ \p ->
      withForeignPtr fp $ \f ->
        memcpy len p (f `plusPtr` off)
```

Pointer Arithmetic The simple body of `copy` abstracts a fair bit of internal work. `memcpy sz dst src`, implemented in C and accessed via the FFI is a potentially dangerous, low-level operation, that copies `sz` bytes starting *from* an address `src` *into* an address `dst`. Crucially, for safety, the regions referred to be `src` and `dst` must be larger than `sz`. We capture this requirement by defining a type alias `PtrGE a N` denoting GHC pointers that refer to a region bigger than `N` bytes, and then specifying that the destination and source buffers for `memcpy` are large enough.

```
type PtrN a N = {v:Ptr a| N <= (plen v)}
memcpy :: sz:CSize -> dst:PtrN a siz
                   -> src:PtrN a siz
                   -> IO ()
```

The actual output for `copy` is created and filled in using the internal function `unsafeCreate` which is a wrapper around

```
create :: l:Nat -> f:(PtrN Word8 l -> IO ())
       -> IO (ByteStringN l)
create l f = do
    fp <- mallocByteString l
    withForeignPtr fp $ \p -> f p
    return $! PS fp 0 l
```

The type of `f` specifies that the action will only be invoked on a pointer of length at least `l`, which is verified by propagating the types of `mallocByteString` and `withForeignPtr`. The fact that the action is only invoked on such pointers is used to ensure that the value `p` in the body of `copy` is of size `l`. This, and the `ByteString` invariant that the size of the payload `fp` exceeds the sum of `off` and `len`, ensures that the call to `memcpy` is safe.

Higher Order Loops `mapAccumR` combines a `map` and a `foldr` over a `ByteString`. The function uses non-trivial recursion, and demonstrates the utility of abstract-interpretation based inference.

```
mapAccumR f z b
  = unSP $ loopDown (mapAccumEFL f) z b
```

To enable fusion [12] `loopDown` uses a higher order `loopWrapper` to iterate over the buffer with a `doDownLoop` action:

```
doDownLoop f acc0 src dest len
  = loop len (len-1) (len-1) acc0
  where
   loop (w::Int) s d acc
     | s < 0
     = return (acc :*: d+1 :*: len - (d+1))
     | otherwise
     = do x <- peekByteOff src s
          case f acc x of
            (acc' :*: NothingS) ->
                 loop (w-1) (s-1) d acc'
            (acc' :*: JustS x') ->
                 pokeByteOff dest d x'
              >> loop (w-1) (s-1) (d-1) acc'
```

The above function iterates across the `src` and `dst` pointers from the right (by repeatedly decrementing the offsets `s` and `d` starting at the high `len` down to `-1`). Low-level reads and writes are carried out using the potentially dangerous `peekByteOff` and `pokeByteOff` respectively. To ensure safety, we type these low level operations with refinements stating that they are only invoked with valid offsets `VO` into the input buffer `p`.

```
type VO P    = {v:Nat | v < plen P}
peekByteOff :: p:Ptr b -> VO p -> IO a
pokeByteOff :: p:Ptr b -> VO p -> a -> IO ()
```

The function `doDownLoop` is an internal function. Via abstract interpretation [30], LIQUIDHASKELL infers that (1) `len` is less than the sizes of `src` and `dest`, (2) `f` (here, `mapAccumEFL`) always returns a `JustS`, so (3) source and destination offsets satisfy $0 \leq \mathtt{s}, \mathtt{d} < \mathtt{len}$, (4) the generated `IO` action returns a triple `(acc :*: 0 :*: len)`, thereby proving the safety of the accesses in `loop` *and* verifying that `loopDown` and the API function `mapAccumR` return a `Bytestring` whose size equals its input's.

To prove *termination*, we add a *witness* `w`. Though `s` decreases at each call, it is *not* a `Nat` as it reaches `-1`. The system infers that `w` decreases and *is* a `Nat` as it equals `s+1`, thus proving termination.

Nested Data Finally, consider `group`, which splits a string like `"aart"` into the list `["aa","r","t"]`, *i.e.* a list of (a) non-empty `ByteString`s whose (b) total length equals that of the input. To specify these requirements, we define a measure for the total length of strings in a list and use it to write an alias for a list of *non-empty* strings whose total length equals that of another string:

```
measure bLens :: [ByteString] -> Int
bLens ([])     = 0
bLens (x:xs)   = bLen x + bLens xs

type ByteStringNE
  = {v:ByteString | bLen v > 0}
type ByteStringsEq B
  = {v:[ByteStringNE] | bLens v = bLen b}
```

LIQUIDHASKELL uses the above to verify that

```
group :: b:ByteString -> ByteStringsEq b
group xs
 | null xs   = []
 | otherwise = let x        = unsafeHead xs
                   xs'      = unsafeTail xs
                   (ys, zs) = spanByte x xs'
               in (y `cons` ys) : group zs
```

The example illustrates why refinements are critical for proving termination. LIQUIDHASKELL determines that `unsafeTail` returns a *smaller* `ByteString` than its input, and that each element returned by `spanByte` is no bigger than the input, concluding that `zs` is smaller than `xs`, and hence checking the body under the termination-weakened environment.

To see why the output type holds, let's look at `spanByte`, which splits strings into a pair:

```
spanByte c ps@(PS x s l)
  = inlinePerformIO $ withForeignPtr x $
        \p -> go l (p `plusPtr` s) 0
  where
    go (w::Int) p i
      | i >= l    = return (ps, empty)
      | otherwise = do
          c' <- peekByteOff p i
          if c /= c'
            then let b1 = unsafeTake i ps
                     b2 = unsafeDrop i ps
                 in  return (b1, b2)
            else go (w-1) p (i+1)
```

Via inference, LIQUIDHASKELL verifies the safety of the pointer accesses, and determines that the sum of the lengths of the output pair of `ByteString`s equals that of the input `ps`. Termination follows by inferring that the sum of the witness `w` and `i` equals `l`.

E.3 Text

The standard Haskell string type is implemented as a list of characters, which makes it easy to reason about but is bad for performance. For serious text-processing Haskellers switch to the `Text`

library, which uses byte arrays and stream fusion to guarantee performance while providing the high-level API.

In our evaluation of LIQUIDHASKELL on `Text` [27], 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. 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 array, and a `length` describing the number of `Word16`s in the `Text`. The `Array` is created and filled using uses a *mutable* `MArray`. All write operations in `Text` are performed on `MArray`s in the `ST` Monad, but they are *frozen* into `Array`s 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

```
measure malen       :: MArray s -> Int
predicate EqLen A MA = alen A = malen MA
predicate Ok I A     = 0 <= I < malen A
type VO A            = {v:Int| Ok v A}

unsafeWrite  :: m:MArray s
             -> VO m -> Word16 -> ST s ()
unsafeFreeze :: m:MArray s
             -> ST s {v:Array | EqLen v m}
```

Reasoning about Unicode The function `writeChar` – originally `Data.Text.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 `Word16`s).

```
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 encoding makes the specification of the function rather interesting. Due to the encoding of `Char`s we cannot just require `i` to be less than the length of `marr`; if `i` were `malen marr - 1` and `c` required two `Word16`s, we would perform an out-of-bounds write. To account for this subtlety, we define a `Room` predicate to ensure that the encoding of `c` is taken into account.

```
measure ord         :: Char -> Int
predicate OkN I A N  = N <= 2 & Ok (I+N-1) A
predicate Room I A C = if ord C <  0x10000
                       then OkN I A 1
                       else OkN I A 2
```

`Room i marr c` can be read as "if `c` is encoded using one `Word16`, then `i` must be less than `malen marr - 1`, otherwise `i` must be less than `malen marr - 2`." Equipped with the above, we define two useful aliases for the specification

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

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

Bug The burden of proving write safety lies with clients of `writeChar`. 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`.

```
mapAccumL f z0 (Stream next0 s0 len) =
  (nz, Text na 0 nl)
 where
  mlen = upperBound 4 len
  (na,(nz,nl)) = runST $ do
       (marr,x) <- (new mlen >>= \arr ->
                    outer arr mlen z0 s0 0)
       arr      <- unsafeFreeze marr
       return (arr,x)
  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
          | otherwise -> do
            let (z',c) = f z x
            d <- writeChar arr i c
            loop z' s' (i+d)
          where j | ord x < 0x10000 = i
                  | otherwise       = i + 1
```

Let's focus on the `Yield x s'` case. We first compute the maximum index `j` to which we will write and determine the safety of a write. If it is safe to write to `j` we call the provided function `f` on the accumulator `z` and the character `x`, and write the *resulting* character `c` into the array. We know nothing about `c` though, specifically whether `c` will be stored as one or two `Word16`s, so LIQUIDHASKELL flags the call to `writeChar` as *unsafe*.

To illustrate why the call is in fact buggy, consider a sample iteration of `loop` where `i = malen arr - 1` and `ord x < 0x10000`. In this case `j` will equal `i` and we will enter the `otherwise` branch. Next, suppose `f z x` returns a `c` such that `ord c >= 0x10000`. The action `writeChar arr i c` will write to indices `i` *and* `i+1` of `arr`, but `i+1 = malen arr` and is not a valid index for writing!

The error lies dormant till the next loop iteration, when `i = malen arr + 1` and we trigger the `j >= top` branch. Here, we allocate a larger array and copy the contents of the previous array into the new array. The `copyM arr' 0 arr 0 top` call only copies `top` elements, *i.e.* it *does not* copy the element *at* `top`, *losing* a `Word16` and so yielding the wrong output. We have reported the error to the authors of the library.