Refinement Types For Haskell *

Niki Vazou Eric L. Seidel Ranjit Jhala
UC San Diego

Dimitrios Vytiniotis Simon Peyton-Jones
Microsoft Research

Abstract

SMT-based checking of refinement types for call-by-value languages is a well-studied subject. Unfortunately, the classical translation of refinement types to verification conditions is unsound under lazy evaluation. When checking an expression, such 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 and the corresponding verification conditions must take into account *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 encode invariants by composing types with SMT-decidable refinement predicates [33, 43], generalizing Floyd-Hoare Logic (*e.g.* EscJava [17]) 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 [32, 43] to functional correctness of data structures [24], security protocols [5], and compiler correctness [37].

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.

We were 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. This would be sound and precise – however it is far from obvious to us how to re-use and extend existing SMT machinery for this purpose. (§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 [12] 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 code (§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

* This work was supported by NSF grants CNS-0964702, CNS-1223850, CCF-1218344, CCF-1018672, and a generous gift from Microsoft Research.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from Permissions@acm.org.

Copyright © ACM [to be supplied]. . . $15.00

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 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 [25, 32] will use the refinement type aliases `Pos` and `Nat` and the specification for `div` from §1 to *accept* `good` and *reject* `bad`. We use the syntax of Figure 1, where r is a *refinement expression*, or just *refinement* for short. We will vary the expressiveness of the language of refinements in different parts of the paper.

```
good     :: Nat -> Nat -> Int
good x y = let z = y + 1 in x `div` z

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

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:

$$\begin{array}{l}\mathtt{x}{:}\{\mathtt{x}{:}\mathtt{Int} \mid \mathtt{x} \geq 0\},\\ \mathtt{y}{:}\{\mathtt{y}{:}\mathtt{Int} \mid \mathtt{y} \geq 0\}\end{array} \vdash \{\mathtt{y}{:}\mathtt{Int} \mid \mathtt{y} \geq 0\} \preceq \{\mathtt{v}{:}\mathtt{Int} \mid \mathtt{v} > 0\}$$

We use the Abbreviations of Figure 1 to simplify the syntax of the queries. So the above query simplifies to:

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

Verification Conditions To discharge the above subtyping query, a refinement type system generates a *verification condition* (VC), a logical formula that stipulates that under the assumptions corresponding to the environment bindings, the refinement in the subtype *implies* the refinement in the super-type. We use the translation $(\!|\cdot|\!)$ shown in Figure 1 to reduce a subtyping query to a verification condition. The translation of a basic type into logic is the refinement of the type. The translation of an environment is the conjunction of its bindings. Finally, the translation of a binding $x{:}\tau$ is the embedding of τ guarded by a predicate denoting that "x is a value". For now, let us ignore this guard and see how the subtyping query for `bad` reduces to the *classical* VC:

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

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 (QF-EUFLIA). Thus, 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, `+`'s type exactly captures its behaviour into the logic:

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

Thus, we can conclude that the divisor `z` is a positive number. The subtyping query for the argument to `div` is

$$\begin{array}{l}\mathtt{x}{:}\{\mathtt{x} \geq 0\}, \mathtt{y}{:}\{\mathtt{y} \geq 0\},\\ \mathtt{z}{:}\{\mathtt{z} = \mathtt{y} + 1\}\end{array} \vdash \{\mathtt{v} = \mathtt{y} + 1\} \preceq \{\mathtt{v} > 0\}$$

Refinements	r	::=	... varies ...
Basic Types	b	::=	$\{v{:}\mathtt{Int} \mid r\} \mid \ldots$
Types	τ	::=	$b \mid x{:}\tau \rightarrow \tau$
Environment	Γ	::=	$\emptyset \mid x{:}\tau, \Gamma$
Subtyping			$\Gamma \vdash \tau_1 \preceq \tau_2$
Abbreviations			
$x{:}\{r\}$		$\doteq$	$x{:}\{x{:}\mathtt{Int} \mid r\}$
$\{x \mid r\}$		$\doteq$	$\{x{:}\mathtt{Int} \mid r\}$
$\{r\}$		$\doteq$	$\{v{:}\mathtt{Int} \mid r\}$
$\{x{:}\{y{:}\mathtt{Int} \mid r_y\} \mid r_x\}$		$\doteq$	$\{x{:}\mathtt{Int} \mid r_x \wedge r_y\,[x/y]\}$
Translation			
$(\!\|\Gamma \vdash b_1 \preceq b_2\|\!)$		$\doteq$	$(\!\|\Gamma\|\!) \Rightarrow (\!\|b_1\|\!) \Rightarrow (\!\|b_2\|\!)$
$(\!\|\{x{:}\mathtt{Int} \mid r\}\|\!)$		$\doteq$	r
$(\!\|x{:}\{v{:}\mathtt{Int} \mid r\}\|\!)$		$\doteq$	"x is a value" $\Rightarrow r\,[x/v]$
$(\!\|x{:}(y{:}\tau_y \rightarrow \tau)\|\!)$		$\doteq$	`true`
$(\!\|x_1{:}\tau_1, \ldots, x_n{:}\tau_n\|\!)$		$\doteq$	$(\!\|x_1{:}\tau_1\|\!) \wedge \ldots \wedge (\!\|x_n{:}\tau_n\|\!)$

Figure 1. Notation: Types, Subtyping & VCs

which reduces to the *valid* VC

$$\begin{array}{l}(\mathtt{x} \geq 0) \wedge (\mathtt{y} \geq 0) \wedge\\ (\mathtt{z} = \mathtt{y} + 1)\end{array} \Rightarrow (\mathtt{v} = \mathtt{y} + 1) \Rightarrow (\mathtt{v} > 0)$$

2.2 Lazy Evaluation Makes VCs Unsound

To generate the classical VC, we ignored the "x is a value" guard that appears in the embedding of a binding $(\!|x{:}\tau|\!)$ (Figure 1). Under lazy evaluation, ignoring this "is a value" guard can lead to unsoundness. 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 [19, 28]. 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

$$\mathtt{n}{:}\{\mathtt{false}\},\ \mathtt{y}{:}\{\mathtt{y} = 0\} \vdash \{\mathtt{v} = 0\} \preceq \{\mathtt{v} > 0\} \qquad (1)$$

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, using the embedding of Figure 1 and ignoring the "is a value" guard, reduces to the VC:

$$\mathtt{false} \wedge \mathtt{y} = 0 \ \Rightarrow (\mathtt{v} = 0) \Rightarrow (\mathtt{v} > 0) \qquad (2)$$

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

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

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" [19]) with VCs is *unsound under lazy evaluation*. To see this, observe that the VC (2) 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 Readers familar with fully dependently typed languages like Cayenne [2], Agda [29], Coq [6], or Idris [8], may be tempted to attribute the unsoundness to the presence of arbitrary recursion and hence non-termination (*e.g.* `diverge`). While it *is* possible to define a sound semantics for dependent types that mention potentially non-terminating expressions [25], it is not clear how to reconcile such semantics with decidable type checking.

Refinement type systems avoid this situation by carefully restricting types so that they do not contain arbitrary terms (even through substitution), but rather only terms from restricted logics that preclude arbitrary user-defined functions [15, 37, 43]. Very much like previous work, we enforce the same restriction with a *well-formedness condition* on refinements (WF-BASE-D in Fig. 6).

However, we show that this restriction *is plainly not sufficient for soundness* when laziness is combined with non-termination, as binders can be bound to diverging expressions. Unsurprisingly, in a strongly normalizing language the question of lazy or strict semantics is irrelevant for soundness, and hence an "easy" way to solve the problem would be to completely eliminate non-termination and rely on the soundness of previous refinement or dependent type systems! Instead, we show here how to recover soundness for a lazy language *without* imposing such a drastic requirement.

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 [25] in which refinements r are *arbitrary* expressions e from the source language. We define a denotation for types and use it to define subtyping declaratively.

Denotations of Types and Environments Recall the type `Pos` defined as `{v:Int | 0 < 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:

$$[\![\{x{:}\tau \mid r\}]\!] \doteq \{e \mid \emptyset \vdash e : \tau, \text{ if } e \hookrightarrow^* w \text{ then } r\,[w/x] \hookrightarrow^* \texttt{true}\}$$

That is, the type denotes the set of expressions e that have the corresponding base type τ which *diverge or* reduce to values that make the refinement reduce to `true`. The guard $e \hookrightarrow^* w$ is crucially required to prove soundness in the presence of recursion. Thus, quoting [25], "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 Subtyping Equipped with interpretations for types and environments, we define the *declarative subtyping* $\preceq$-BASE (over basic types b, shown in Figure 1) to be containment between the types' denotations:

$$\frac{\forall \theta \in [\![\Gamma]\!].\ [\![\theta(\{v{:}B \mid r_1\})]\!] \subseteq [\![\theta(\{v{:}B \mid r_2\})]\!]}{\Gamma \vdash \{v{:}B \mid r_1\} \preceq \{v{:}B \mid r_2\}} \preceq\text{-BASE}$$

Let us revisit the `explode` example from §2.2; recall that the function is safe under eager evaluation but unsafe under lazy evaluation. Let us see how the declarative subtyping allows us to reject in the one case and accept in the other.

Declarative Subtyping with Lazy Evaluation Let us revisit the query (1) to see whether it holds under the declarative subtyping rule $\preceq$-BASE. The denotation containment

$$\forall \theta \in [\![\texttt{n}{:}\{\texttt{false}\},\ \texttt{y}{:}\{\texttt{y} = \texttt{0}\}]\!].[\![\theta\ \{\texttt{v} = \texttt{0}\}]\!] \subseteq [\![\theta\ \{\texttt{v} > \texttt{0}\}]\!] \quad (3)$$

does not hold. To see why, consider a θ that maps n to any diverging expression of type `Int` and y to the value 0. Then, $0 \in [\![\theta\ \{\texttt{v} = \texttt{0}\}]\!]$ but $0 \notin [\![\theta\ \{\texttt{v} > \texttt{0}\}]\!]$, thereby showing that the denotation containment does not hold.

Declarative Subtyping with Eager Evaluation Since denotational containment (3) does not hold, 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 (3) 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 containment (3) 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 (*i.e.* under call-by-value) reduction from subtyping to VC is sound.

Algorithmic Subtyping via Verification Conditions The above subtyping ($\preceq$-BASE) rule allows us to prove preservation and progress [25] but quantifies over evaluation of arbitrary expressions, and so is undecidable. To make checking *algorithmic* we approximate the denotational containment using *verification conditions* (VCs), formulas drawn from a decidable logic, that are valid only if the undecidable containment 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 subtyping, to obtain both sound and 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 and linear arithmetic (QF-EUFLIA). We design our typing rules to ensure that for any valid derivation, all the refinements belong in this restricted language.

Step 2: Translating Containment into VCs Our goal is to encode the denotation containment antecedent of $\preceq$-BASE

$$\forall \theta \in [\![\Gamma]\!].\ [\![\theta(\{v{:}B \mid r_1\})]\!] \subseteq [\![\theta(\{v{:}B \mid r_2\})]\!] \quad (4)$$

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. We use the variable x to approximate denotational containment by stating that if x belongs to the type $\{v{:}B \mid r_1\}$ then x belongs to the type $\{v{:}B \mid r_2\}$:

$$\forall X \in dom(\Gamma), x.(\!|\Gamma|\!) \Rightarrow (\!|x{:}\{v{:}B \mid r_1\}|\!) \Rightarrow (\!|x{:}\{v{:}B \mid r_2\}|\!)$$

where $(\!|\Gamma|\!)$ and $(\!|x{:}\tau|\!)$ are respectively the translation of the environment and bindings into logical formulas that are only satisfied

by assignments $(\!|\theta|\!)$ as shown in Figure 1. Using the translation of bindings, and by renaming x to v, we rewrite the the condition as

$$\begin{aligned}\forall X \in dom(\Gamma), v.(\!|\Gamma|\!) \quad &\Rightarrow (\text{“}v \text{ is a value”} \Rightarrow r_1) \\ &\Rightarrow (\text{“}v \text{ is a value”} \Rightarrow r_2)\end{aligned}$$

Type refinements are carefully chosen to belong to the decidable logical sublanguage QF-EUFLIA, thus we directly translate type refinements into the logic. Thus, what is left is to translate into logic the environment and the “is a value” guards. We postpone translation of the guards as we approximate the above formula by a *stronger*, *i.e.* sound with respect to 4, VC that just omits the guards:

$$\forall X \in dom(\Gamma), v.(\!|\Gamma|\!) \Rightarrow r_1 \Rightarrow r_2$$

To translate environments, we conjoin their bindings' translations:

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

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

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

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

The crucial question is: *how* can one discharge these guards to conclude that x indeed satisfies r? 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, which fall 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 (2) 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*. We use a non-deterministic β-reduction (formally defined in §3) to label each type 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, in a finite number of steps, to *Haskell values* in Weak Head Normal Form (WHNF). Up to now we only discussed `Int` basic types, but our theory supports user-defined algebraic data types. An expression like `0 : repeat 0` is an infinite Haskell value. As we shall discuss, such infinite values cannot be represented in the logic. To distinguish infinite from finite values, we use a Fin-type, written $\tau^{\Downarrow}$, to label binders of expressions that are guaranteed to reduce to *finite values* with no redexes. This stratification lets us generate VCs that are sound for lazy evaluation. Let B be a basic labelled type. The key piece is the translation of environment bindings:

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

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 (1) yields the *invalid* VC

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

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

As binders appear in refinements, and binders may refer to potentially infinite computations (*e.g.* `[0..]`), we must ensure that refinements are well defined (*i.e.* do not diverge). We achieve this via stratification itself, *i.e.* by ensuring that all refinements have type $\texttt{Bool}^{\Downarrow}$. By Corollary 1, this suffices to ensure that all the refinements are indeed well-defined and converge.

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. Finally, we use the termination checker to ensure that all refinements are well-defined (*i.e.* do converge.)

Example: 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:

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

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

Example: Improving Precision By Forcing Evaluation 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:

```
ex2 = 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 `ex2` is obviously safe, but the system would reject it, as the call yields the subtyping query:

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

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

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

We could solve the problem by forcing evaluation of x. In Haskell the seq operator or a bang-pattern can be used to force evaluation. In our system the same effect is achieved by the **case-of** primitive: inside each case the matched binder is guaranteed to be a Haskell value in WHNF. This intuition is formalized by the typing rule (T-CASE-D), which checks each case after assuming the scrutinee and the match binder have Wnf types.

If we force x's evaluation, using the case primitive, the call to div yields the subtyping query:

$$\begin{array}{l}\texttt{x}{:}\{\texttt{x}{:}\texttt{Int}^{\downarrow} \mid \texttt{x} = 1\} \\ \texttt{y}{:}\{\texttt{y}{:}\texttt{Int} \mid \texttt{y} > \texttt{x}\}\end{array} \vdash \{\texttt{v} > \texttt{x}\} \preceq \{\texttt{v} > 0\} \quad (5)$$

As x is Wnf, we accept ex2 by proving the validity of the VC

$$\texttt{x} = 1 \Rightarrow \texttt{v} > \texttt{x} \Rightarrow \texttt{v} > 0 \quad (6)$$

Example: Improving Precision By Termination While forcing evaluation allows us to ensure that certain environment binders have non-Div types, it requires program rewriting using case-splitting or the seq operator which leads to 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, in the ex2 example the term 1 obviously terminates. Hence, we type x as $\texttt{Int}^{\Downarrow}$, yielding the subtyping query for div application:

$$\begin{array}{l}\texttt{x}{:}\{\texttt{x}{:}\texttt{Int}^{\Downarrow} \mid \texttt{x} = 1\} \\ \texttt{y}{:}\{\texttt{y}{:}\texttt{Int} \mid \texttt{y} > \texttt{x}\}\end{array} \vdash \{\texttt{v} > \texttt{x}\} \preceq \{\texttt{v} > 0\} \quad (7)$$

As x is Fin, we accept ex2 by proving the validity of the VC

$$\texttt{x} = 1 \Rightarrow \texttt{v} > \texttt{x} \Rightarrow \texttt{v} > 0 \quad (8)$$

Example: 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 (7) and VC (8), 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 [38] as embodied in sized types [20, 42]. 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 (using terminating fixpoints of §4.2.) For example, to prove f terminates, we check its body in an environment

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

where we have weakened the type of f to stipulate that it *only* be (recursively) called with Nat values n′ that are *strictly less than* the (current) parameter n. The argument of f exactly captures these constraints, as using the Abbreviations of Figure 1 the argument of f is expanded to $\{\texttt{n}'{:}\texttt{Int}^{\Downarrow} \mid \texttt{n}' < \texttt{n} \wedge \texttt{n}' >= 0\}$. The body typechecks as the recursive call generates the valid VC

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

Definition

$$def \quad ::= \quad \texttt{measure}\ f :: \tau \\ \qquad\qquad eq_1 \ldots eq_n$$

Equation

$$eq \quad ::= \quad f\ (D\ \overline{x}) = r$$

Equation to Type

$$(\!|f\ (D\ \overline{x}) = r|\!) \quad \doteq \quad D :: \overline{x{:}\tau} \rightarrow \{\texttt{v}{:}\tau \mid f\ \texttt{v} = r\}$$

Figure 2. Syntax of Measures

Example: Diverging Refinements In this final example we discuss why refinements should always converge and how we statically ensure convergence. Consider the invalid specification

```
diverge 0 :: {v:Int | v = 12}
```

that states that the value of a diverging integer is 12. The above specification should be rejected, as the refinement $\texttt{v} = 12$ does not evaluate to true ($\texttt{diverge}\ 0 = 12 \not\hookrightarrow^{*} \texttt{true}$), instead it diverges.

We want to check the validity of the formula $\texttt{v} = 12$ under a model that maps v to the diverging integer diverge 0. Any system that decides this formula to be true will be unsound, *i.e.* the VCs will not soundly approximate subtyping. For similar reasons, the system should not decide that this formula is false. To reason about diverging refinements one needs three valued logic, where logical formulas can be solved to true, false, or diverging. Since we want to discharge VC using SMT solvers that currently do not support three valued reasoning, we exclude diverging refinements from types. To do so, we restrict $=$ to finite integers

$$= :: \texttt{Int}^{\Downarrow} \rightarrow \texttt{Int}^{\Downarrow} \rightarrow \texttt{Bool}^{\Downarrow}$$

and we say that $\{v{:}B \mid r\}$ is well-formed *iff* r has a $\texttt{Bool}^{\Downarrow}$ type (Corollary 1). Thus the initial invalid specification will be rejected as non well-formed.

2.6 Measures: From Integers to Data Types

So far, all our examples have used only integer and boolean expressions in refinements. To describe properties of algebraic data types, we use *measures*, introduced in prior work on Liquid Types [24]. Measures are inductively defined functions that can be used in refinements, and provide an efficient way to axiomatize properties of data types. For example, emp determines whether a list is empty:

```
measure emp  :: [Int] -> Bool
  emp []     = true
  emp (x:xs) = false
```

The syntax for measures deliberately looks like Haskell, but it is *far* more restricted, and should really be considered as a separate language. A measure has exactly one argument, and is defined by a list of equations, each of which has a simple pattern on the left hand side (see Figure 2). The right-hand side of the equation is a refinement expression r. Measure definitions are typechecked in the usual way; we omit the typing rules which are standard. (Our metatheory does not support type polymorphism, so in this paper we simply reason about lists of integers; however, our implementation supports polymorphism.)

Denotational semantics The denotational semantics of types in λ^H in §2.3 is readily extended to support measures. In λ^H a refinement r is an arbitrary expression, and calls to a measure are evaluated in the usual way by pattern matching. For example, with the above definition of emp it is straightforward to show that

$$[1, 2, 3] :: \{\texttt{v}{:}[\texttt{Int}] \mid \texttt{not}\ (\texttt{emp}\ \texttt{v})\} \quad (9)$$

as the refinement `not (emp ([1, 2, 3]))` evaluates to `true`.

Measures as Axioms How can we reason about invocations of measures in the decidable logic of VCs? A natural approach is to treat a measure like `emp` as an uninterpreted function, and add logical axioms that capture its behaviour. This looks easy: each equation of the measure definition corresponds to an axiom, thus:

$$\texttt{emp}\ [] = \texttt{true}$$
$$\forall \texttt{x}, \texttt{xs}.\ \texttt{emp}\ (\texttt{x} : \texttt{xs}) = \texttt{false}$$

Under these axioms the judgement 9 is indeed valid.

Measures as Refinements in Types of Data Constructors Axiomatizing measures is *precise*; that is, the axioms exactly capture the meaning of measures. Alas, axioms render SMT solvers *inefficient*, and render the VC mechanism *unpredictable*, as one must rely on various brittle syntactic matching and instantiation heuristics [14].

Instead, we use a different approach that is *both* precise *and* efficient. The key idea is this: *instead of translating each measure equation into an axiom, we translate each equation into a refined type for the corresponding data constructor* [24]. This translation is given in Figure 2. For example, the definition of the measure `emp` yields the following refined types for the list data constructors:

$$\begin{array}{lll} [] & :: & \{\texttt{v}{:}[\texttt{Int}] \mid \texttt{emp v} = \texttt{true}\} \\ : & :: & \texttt{x}{:}\texttt{Int} \rightarrow \texttt{xs}{:}[\texttt{Int}] \rightarrow \{\texttt{v}{:}[\texttt{Int}] \mid \texttt{emp v} = \texttt{false}\} \end{array}$$

These types ensure that: (1) each time a list value is *constructed*, its type carries the appropriate emptiness information. Thus our system is able to statically decide that (9) is valid, and, (2) each time a list value is *matched*, the appropriate emptiness information is used to improve precision of pattern matching, as we see next.

Using Measures As an example, we use the measure `emp` to provide an appropriate type for the `head` function:

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

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

`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. The call to `error` generates the subtyping query

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

The match-binder `b` holds the result of the match [36]. In the `[]` case, we assign it the refinement of the type of `[]` which is $(\texttt{emp xs}) = \texttt{true}$. Since the call is done inside a **`case`**–**`of`** expressions both `xs` and `b` are guaranteed to be in WHNF, thus they have Wnf types.

The verifier *accepts* the program as the above subtyping reduces to the valid VC

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

Consequently, our system can naturally support idiomatic Haskell, *e.g.* taking the `head` of an infinite list:

```
ex x      = head (repeat x)

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

Multiple Measures If a type has multiple measures, we simply refine each data constructor's type with the *conjunction* of the refinements from each measure. For example, consider a measure that computes the length of a list:

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

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 3. λ^U: Syntax and Operational Semantics

```
measure len  :: [Int] -> Int
  len ([])   = 0
  len (x:xs) = 1 + len xs
```

Using the translation of Figure 2, we extract the following types for list's data constructors.

$$\begin{array}{lll} [] & :: & \{\texttt{v}{:}[\texttt{Int}] \mid \texttt{len v} = 0\} \\ : & :: & \texttt{x}{:}\texttt{Int} \rightarrow \texttt{xs}{:}[\texttt{Int}] \rightarrow \{\texttt{v}{:}[\texttt{Int}] \mid \texttt{len v} = 1 + (\texttt{len xs})\} \end{array}$$

The final types for list data constructors will be the conjunction of the refinements from `len` and `emp`:

$$\begin{array}{lll} [] & :: & \{\texttt{v}{:}[\texttt{Int}] \mid \texttt{emp v} = \texttt{true} \wedge \texttt{len v} = 0\} \\ : & :: & \texttt{x}{:}\texttt{Int} \rightarrow \texttt{xs}{:}[\texttt{Int}] \rightarrow \\ & & \{\texttt{v}{:}[\texttt{Int}] \mid \texttt{emp v} = \texttt{false} \wedge \texttt{len v} = 1 + (\texttt{len xs})\} \end{array}$$

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-EUFLIA, 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.

3.1 Syntax

Figure 3 summarizes the syntax of λ^U, which is essentially the calculus λ^H [25] *without* the dynamic checking features (like casts), but *with* the addition of data constructors. In λ^U, as in λ^H, refinement expressions r are not drawn from a decidable logical sublanguage, but can be arbitrary expressions e (hence $r ::= e$ in Figure 3). This choice allows us to prove preservation and progress, but renders typechecking undecidable.

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 $[\texttt{Int}]$, which represents lists of integers, has two data constructors: [] and :, *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 3 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 $[\texttt{Int}]$ 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* $x{:}\tau_x \to \tau$ 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 $\tau_x \to \tau$ to abbreviate $x{:}\tau_x \to \tau$ if x does not appear in τ. We use $_$ for unused binders. We write $\{v{:}\texttt{nat}^l \mid r\}$ to abbreviate $\{v{:}\texttt{Int}^l \mid 0 \leq v \wedge r\}$.

Denotations Each type τ *denotes* a set of expressions $[\![\tau]\!]$, that are defined via the dynamic semantics [25]. 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:

$$[\![\{x{:}B \mid r\}]\!] \doteq \{e \mid e{:}B,\ \text{if } e \hookrightarrow^* w \text{ then } r\,[w/x] \hookrightarrow^* \texttt{true}\}$$
$$[\![x{:}\tau_x \to \tau]\!] \doteq \{e \mid e{:}\lfloor\tau_x \to \tau\rfloor, \forall e_x \in [\![\tau_x]\!].\ e\ e_x \in [\![\tau\,[e_x/x]]\!]\}$$

Constants For each constant c we define its type $\mathsf{Ty}(c)$ such that $c \in [\![\mathsf{Ty}(c)]\!]$. For example,

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

So, by definition we get the constant typing lemma

Lemma 1. *[Constant Typing] Every constant* $c \in [\![\mathsf{Ty}(c)]\!]$.

Thus, if $\mathsf{Ty}(c) \doteq x{:}\tau_x \to \tau$, then for every value $w \in [\![\tau_x]\!]$, we require that $\delta(c, w) \in [\![\tau\,[w/x]]\!]$. For every value $w \notin [\![\tau_x]\!]$, it suffices to define $\delta(c, w)$ 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, as discussed in §2.6 we use `emp` to refine the list data constructors' types:

$$\begin{array}{lcl}
\mathsf{Ty}([]) & \doteq & \{\texttt{v:[Int]} \mid \texttt{emp v}\} \\
\mathsf{Ty}(:) & \doteq & \texttt{Int} \to [\texttt{Int}] \to \{\texttt{v:[Int]} \mid \neg(\texttt{emp v})\}
\end{array}$$

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

$$\frac{\Gamma, v{:}B \vdash_U r : \texttt{Bool}}{\Gamma \vdash_U \{v{:}B \mid r\}}\ \textsc{Wf-Base} \qquad \frac{\Gamma \vdash_U \tau_x \quad \Gamma, x{:}\tau_x \vdash_U \tau}{\Gamma \vdash_U x{:}\tau_x \to \tau}\ \textsc{Wf-Fun}$$

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

$$\frac{\forall \theta \in [\![\Gamma]\!].[\![\theta(\{v{:}B \mid r_1\})]\!] \subseteq [\![\theta(\{v{:}B \mid r_2\})]\!]}{\Gamma \vdash_U \{v{:}B \mid r_1\} \preceq \{v{:}B \mid r_2\}}\ \preceq\textsc{-Base}$$

$$\frac{\Gamma \vdash_U \tau'_x \preceq \tau_x \quad \Gamma, x{:}\tau'_x \vdash_U \tau \preceq \tau'}{\Gamma \vdash_U x{:}\tau_x \to \tau \preceq x{:}\tau'_x \to \tau'}\ \preceq\textsc{-Fun}$$

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

$$\frac{(x, \tau) \in \Gamma}{\Gamma \vdash_U x : \tau}\ \textsc{T-Var} \qquad \frac{}{\Gamma \vdash_U c : \mathsf{Ty}(c)}\ \textsc{T-Con}$$

$$\frac{\Gamma \vdash_U e : \tau' \quad \Gamma \vdash_U \tau' \preceq \tau \quad \Gamma \vdash_U \tau}{\Gamma \vdash_U e : \tau}\ \textsc{T-Sub}$$

$$\frac{\Gamma, x{:}\tau_x \vdash_U e : \tau \quad \Gamma \vdash_U \tau_x}{\Gamma \vdash_U \lambda x.e : (x{:}\tau_x \to \tau)}\ \textsc{T-Fun}$$

$$\frac{\Gamma \vdash_U e_1 : (x{:}\tau_x \to \tau) \quad \Gamma \vdash_U e_2 : \tau_x}{\Gamma \vdash_U e_1\ e_2 : \tau\,[e_2/x]}\ \textsc{T-App}$$

$$\frac{\Gamma \vdash_U e_x : \tau_x \quad \Gamma, x{:}\tau_x \vdash_U e : \tau \quad \Gamma \vdash_U \tau}{\Gamma \vdash_U \texttt{let}\ x = e_x\ \texttt{in}\ e : \tau}\ \textsc{T-Let}$$

$$\frac{\begin{array}{c}\Gamma \vdash_U e : \{v{:}T \mid r\} \qquad \Gamma \vdash_U \tau \\ \forall i. \mathsf{Ty}(D_i) = \overline{y_j{:}\tau_j} \to \{v{:}T \mid r_i\} \\ \Gamma, \overline{y_j{:}\tau_j}, x{:}\{v{:}T \mid r \wedge r_i\} \vdash_U e_i : \tau\end{array}}{\Gamma \vdash_U \texttt{case}\ x = e\ \texttt{of}\ \{D_i\ \overline{y_j} \to e_i\} : \tau}\ \textsc{T-Case}$$

Figure 4. Type-checking for λ^U

By construction it is easy to prove that Lemma 1 holds for data constructors. For example, `emp` [] goes to `true`.

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 three kinds of rules: Well-formedness, Subtyping, and Typing [5, 25]. A judgment $\Gamma \vdash_U \tau$ 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 $\Gamma \vdash_U \tau_1 \preceq \tau_2$ 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 denotational containment checking. That is, for any closing substitution θ in the denotation of Γ, for every expression e, if $e \in [\![\theta(\tau_1)]\!]$ then $e \in [\![\theta(\tau_2)]\!]$. A judgment $\Gamma \vdash_U e : \tau$ states that the expression e has the type τ in the environment Γ. That is, when the free vari-

Expressions, ***Values***, ***Constants***, ***Basic types***: see Figure 3

Types	τ	::=	$\{v{:}B \mid r\} \mid \{v{:}B^l \mid r\}$
		$\mid$	$x{:}\tau \to \tau$
Labels	l	::=	$\downarrow \mid \Downarrow$
Refinements	r	::=	p
Predicates	p	::=	$p = p \mid p < p \mid p \wedge p \mid \neg p$
		$\mid$	$n \mid x \mid f\ \overline{p} \mid p \oplus p$
		$\mid$	$\texttt{true} \mid \texttt{false}$
Measures	f, g, h		
Operators	$\oplus$	::=	$+ \mid - \mid \ldots$
Integers	n	::=	$0 \mid 1 \mid -1 \mid \ldots$
Domain	d	::=	$n \mid c_w \mid D\ \overline{d} \mid \texttt{true} \mid \texttt{false}$
Model	σ	::=	$x_1 \mapsto d_1, \ldots, x_n \mapsto d_n$
Lifted Values	$w^\perp$	::=	$c \mid \lambda x.e \mid D\ \overline{w^\perp} \mid \perp$

Figure 5. Syntax of λ^D

ables in e are bound to expressions described by Γ, the expression e will evaluate to a value described by τ.

Soundness Following λ^H [25], we use the (undecidable) $\preceq$-BASE 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 $\emptyset \vdash_U e : \tau$ and $e \hookrightarrow e'$, then $\emptyset \vdash_U e' : \tau$.
- **Progress:** If $\emptyset \vdash_U e : \tau$ and $e \neq w$, 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 $\emptyset \vdash_U e : \tau$, $e \hookrightarrow^* w$ then $\emptyset \vdash_U w : \tau$.*
- ***Crash-Freedom:*** *If $\emptyset \vdash_U e : \tau$ then $e \not\hookrightarrow^* \texttt{crash}$.*

We prove the above following the overall recipe of [25]. 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 2. *[Denotation Typing] If $\emptyset \vdash_U e : \tau$ then $e \in [\![\tau]\!]$.*

4. Algorithmic Typing: λ^D

While λ^U is sound, it cannot be *implemented* thanks to the undecidable denotational containment rule $\preceq$-BASE (Figure 4). Next, we go from λ^U to λ^D, a core calculus with sound, SMT-based algorithmic type-checking in four steps. First, we show how to restrict the language of refinements to an SMT-decidable sub-language QF-EUFLIA (§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-EUFLIA and the stratification to approximate the undecidable $\preceq$-BASE with a decidable verification condition $\preceq$-BASE-D, thereby obtaining the algorithmic system λ^D (§4.3).

4.1 Refinement Logic: QF-EUFLIA

Figure 5 summarizes the syntax of λ^D. Refinements r are now predicates p, drawn from QF-EUFLIA, the decidable logic of equality, uninterpreted functions and linear arithmetic [27]. Predicates p include linear arithmetic constraints, function application

All rules as in Figure 4 except as follows:

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

$$\frac{\Gamma, v{:}B \vdash_D p : \texttt{Bool}^{\Downarrow}}{\Gamma \vdash_D \{v{:}B \mid p\}}\ \text{WF-BASE-D}$$

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

$$\frac{(\!|\Gamma, v : B|\!) \Rightarrow (\!|p_1|\!) \Rightarrow (\!|p_2|\!) \text{ is valid}}{\Gamma \vdash_D \{v{:}B \mid p_1\} \preceq \{v{:}B \mid p_2\}}\ \preceq\text{-BASE-D}$$

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

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

$$\frac{\begin{array}{c} l \notin \{\Downarrow, \downarrow\} \Rightarrow \tau \text{ is Div} \\ \Gamma \vdash_D e : \{v{:}T^l \mid r\} \qquad \Gamma \vdash_D \tau \\ \forall i. \mathsf{Ty}(D_i) = \overline{y_j}\tau_j \to \{v{:}T \mid r_i\} \\ \Gamma, \overline{y_j{:}\tau_j}, x{:}\{v{:}T^{\downarrow} \mid r \wedge r_i\} \vdash_D e_i : \tau \end{array}}{\Gamma \vdash_D \texttt{case}\ x = e\ \texttt{of}\ \{D_i\ \overline{y_j} \to e_i\} : \tau}\ \text{T-CASE-D}$$

Figure 6. Typechecking for λ^D

where function symbols correspond to measures (as described in §2.6), and boolean combinations of sub-predicates.

Well-Formedness For a predicate to be well-formed it should be boolean and arithmetic operators should be applied to integer terms, measures should be applied to appropriate arguments (*i.e.* `emp` is applied to [`Int`]), 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 we require that each refinement r has type $\texttt{Bool}^{\Downarrow}$ (rule WF-BASE-D in Figure 6).

Assignments Figure 5 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_w for each value w 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 boolean predicates in the logic over the domain $\mathcal{D}$. We write $\sigma \models p$ if σ is a model of p. We omit the formal definition for space. A predicate p is *satisfiable* if there *exists* $\sigma \models p$. A predicate p is *valid* if *for all* assignments $\sigma \models p$.

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* $w^\perp$ (Figure 5), 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 $\emptyset \vdash_D \theta^\perp(p) : \texttt{Bool}^{\Downarrow}$, 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 $\preceq$-BASE so that only predicates from the decidable logic QF-EUFLIA (not arbitrary expressions) appear in implications $(\!|\Gamma|\!) \Rightarrow$

$\{v{:}b \mid p_1\} \Rightarrow \{v{:}b \mid p_2\}$. Towards this goal, as shown in Figures 5 and 6, 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-D to applications to variables, which ensures that refinements remain within the logic after substitution [32]. Recall, that this is not enough to ensure that refinements do converge, as under lazy evaluation, even binders can refer to potentially divergent values.

4.2 Stratified Types

The typing rules for λ^D are given in Figure 6. 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 2, 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 (using β-reduction as defined in Figure 3) to a value w (resp. *finite* value, *i.e.* an element of the inductively defined $\mathcal{D}$). Formally,

$$\textbf{\textit{Wnf types}} \quad [\![\{v{:}B^{\downarrow} \mid r\}]\!] \doteq [\![\{v{:}B \mid r\}]\!] \cap \{e \mid e \hookrightarrow^* w\} \quad (10)$$

$$\textbf{\textit{Fin types}} \quad [\![\{v{:}B^{\Downarrow} \mid r\}]\!] \doteq [\![\{v{:}B \mid r\}]\!] \cap \{e \mid e \hookrightarrow^* d\} \quad (11)$$

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

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

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-D 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 6, remain the same as in Figure 4. 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).

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

It is easy to prove that the above primitives respect their stratification labels, *i.e.* belong in the denotations of their types.

Note that the above types are restricted in that they can only be applied to finite arguments. In future work, we could address this issue with unrefined versions of primitive types that soundly allow operation on arbitrary arguments. For example, with the current type for $+$, addition of potentially diverging expressions is rejected. Thus, we could define an unrefined signature

$$\mathsf{Ty}(+) \doteq x{:}\texttt{Int} \rightarrow y{:}\texttt{Int} \rightarrow \texttt{Int}$$

and allow the two types of $+$ to co-exist (as an intersection type), where the type checker would choose the precise refined type if and only if both of $+$'s arguments are finite.

Diverging Fixpoints ($\texttt{fix}_\tau$) Next, note that the only place where divergence enters the picture is through the fixpoint combinators used to encode recursion. For any function or basic type $\tau \doteq \tau_1 \rightarrow \ldots \rightarrow \tau_n$, we define the *result* to be the type τ_n.

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

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

i.e., $\texttt{fix}_\tau$ yields recursive functions of type τ. Of course, $\texttt{fix}_\tau$ belongs in the denotation of its type [31] *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 $\texttt{tfix}^n_\tau$:

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

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:

$$\begin{aligned}
\delta(\texttt{tfix}_\tau, f) &\doteq \lambda n.f\ n\ (\texttt{tfix}^n_\tau\ f) \\
\mathsf{Ty}(\texttt{tfix}_\tau) &\doteq (n{:}\texttt{nat}^{\Downarrow} \rightarrow \tau_n \rightarrow \tau) \rightarrow \texttt{nat}^{\Downarrow} \rightarrow \tau
\end{aligned}$$

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:

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

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

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

To understand *why*, note that $\texttt{tfix}^n_\tau$ is only called with arguments strictly smaller than n

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

Soundness of Stratification To formally *prove* that stratification is soundly enforced, it suffices to prove that the Denotation Lemma 2 holds for λ^D. This, in turn, boils down to proving that each (stratified) constant belongs in its type's denotation, *i.e.* each $c \in [\![\mathsf{Ty}(c)]\!]$

or that the Lemma 1 holds for λ^D. The crucial part of the above is proving that the indexed and terminating fixpoints inhabit their types' denotations.

Theorem 3. *[Fixpoint Typing]*

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

With the above we can prove soundness of Stratification as a corollary Denotation Lemma 2, given the interpretations of the stratified types.

Corollary 1. *[Soundness of Stratification]*

1. *If $\emptyset \vdash_D e : \tau^{\Downarrow}$, then evaluation of e is finite.*
2. *If $\emptyset \vdash_D e : \tau^{\downarrow}$, then e reduces to WHNF.*
3. *If $\emptyset \vdash_D e : \{v{:}\tau \mid p\}$, then p cannot diverge.*

Finally, as a direct implication the well-formedness rule WF-BASE-D we conclude 3, *i.e.* that refinements cannot diverge.

4.3 Verification With Stratified Types

We can put the pieces together to obtain an algorithmic implication rule $\preceq$-BASE-D instead of the undecidable $\preceq$-BASE (from Figure 4). Intuitively, each closing substitution θ will correspond to a set of logical assignments $(\!|\theta|\!)$. Thus, we will translate Γ into logical formula $(\!|\Gamma|\!)$ and denotation inclusion into logical implication such that:

- $\theta \in [\![\Gamma]\!]$ iff all $\sigma \in (\!|\theta|\!)$ satisfy $(\!|\Gamma|\!)$, and
- $\theta\{v{:}B \mid p_1\} \subseteq \theta\{v{:}B \mid p_2\}$ iff all $\sigma \in (\!|\theta|\!)$ satisfy $p_1 \Rightarrow p_2$.

Translating Refinements & Environments To translate environments into logical formulas, 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

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

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. (10), (11)) 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 $\preceq$-BASE-D approximates the undecidable $\preceq$-BASE.

Theorem 4. *If $(\!|\Gamma|\!) \Rightarrow p_1 \Rightarrow p_2$ is valid then*

$$\Gamma \vdash_U \{v{:}B \mid p_1\} \preceq \{v{:}B \mid p_2\}$$

To prove the above, let $VC \doteq (\!|\Gamma|\!) \Rightarrow p_1 \Rightarrow p_2$. We prove that if the VC is valid then $\Gamma \vdash_U \{v{:}b \mid p_1\} \preceq \{v{:}b \mid p_2\}$. This 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 2 to prove that the validity of the VC demonstrates the denotational containment $\forall\theta \in [\![\Gamma]\!].[\![\theta(\{v{:}B \mid p_1\})]\!] \subseteq [\![\theta(\{v{:}B \mid p_2\})]\!]$. The soundness of algorithmic typing follows from Theorems 4 and 1:

Theorem 5. *[Soundness of λ^D]*

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

To prove approximation we need to prove that Lemma 1 holds for each constant, and thus it holds for data constructors. In the metatheory we assume a stronger notion of validity that respects the measure axioms. However, since our implementation does not use axioms and instead, without loss of soundness or precision, treats measures as uninterpreted during SMT validity checking, we omit further discussion of axioms for clarity.

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 [36] 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 $\texttt{fac} : \{\texttt{v:nat}^{\Downarrow} \mid \texttt{v} < \texttt{n}\} \to \texttt{nat}^{\Downarrow}$ The recursive call generates the subtyping query:

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

Which reduces to the valid VC

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

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 [18].

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 n | n == 0    = x
         | otherwise = tfac (n*x) (n-1)
```

We specify that the *last parameter* is decreasing by specifying an explicit termination metric `/ [n]` in the type signature. LIQUIDHASKELL *desugars* the termination metric into a new `nat`-valued *ghost parameter* `d` whose value is always equal to the termination metric `n`:

```
tfac :: d:Nat -> Nat -> {n:Nat | d = n} -> Nat
tfac d x n | n == 0    = x
           | otherwise = tfac (n-1) (n*x) (n-1)
```

Type checking, as before, checks the body in an environment where the first argument of `tfac` is weakened, *i.e.*, requires proving `d > n-1`. So, the system needs to know that the ghost argument `d` represents the decreasing metric. We capture this information in the type signature of `tfac` where the *last* argument exactly specifies that `d` is the termination metric `n`, *i.e.*, `d = n`. Note that since the termination metric can depend on any argument, it is important to refine the last argument, so that all arguments are in scope, with the fact that `d` is the termination metric.

To generalize, desugaring of termination metrics proceeds as follows. Let f be a recursive function with parameters $\overline{x}$, and termination metric $\mu(\overline{x})$. Then LIQUIDHASKELL will

- add a `nat`-valued ghost first parameter d in the definition of f,
- weaken the last argument of f with the refinement $d = \mu(\overline{x})$,
- at each recursive call of $f\ \overline{e}$, apply $\mu(\overline{e})$ as the first argument.

Explicit Termination Expressions Let us now apply the previous technique in a function where none of the parameters themselves decrease across recursive calls, but there is some *expression* that forms the decreasing metric. Consider `range lo hi`, which returns the list of `Int`s from `lo` to `hi`: 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:Nat -> {hi:Nat | hi >= lo} -> [Nat]
       / [hi-lo]
range lo hi
  | lo < hi = lo : range (lo + 1) hi
  | _       = []
```

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 (hi-lo) lo hi
  where
    go :: d:Nat -> lo:Nat
       -> {hi:Nat | d = hi - lo} -> [Nat]
    go d lo hi
     | lo < hi = l : go (hi-(lo+1)) (lo+1) hi
     | _       = []
```

After which, it proves `go` terminating, by showing that the first argument `d` 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 in fact 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.

Let us see how we can use refinements to statically distinguish between finite and infinite streams. The direct, *global* route of using a measure

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

to describe infinite lists is unavailable as such a measure, and hence, the corresponding refinement would be non-terminating. Instead, we describe infinite lists in *local* fashion, by stating that each *tail* is non-empty.

Step 1: Abstract Refinements We can parametrize a datatype with abstract refinements that relate sub-parts of the structure [39]. 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 -> L 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.

5.3 User Specifications and Type Inference

In program verification it is common that the user provides functional specification that the code should satisfy. In LIQUIDHASKELL these specifications can be provided as type signatures for **`let`**-bound variables. Consider the typechecking rules of Figure 4 that is used by λ^D.

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

Note that T-LET *guesses* an appropriate type τ_x for e_x and binds this type to x to typecheck e.

LIQUIDHASKELL allows the user to specify the type τ_x for top level bindings. For every binding $\mathtt{let}\ x = e_x\ \mathtt{in}\ \ldots$, if the user provides a type specification τ_x, LIQUIDHASKELL checks using the appropriate environment (1) that the specified type is well-formed, and (2) that the expression e_x typechecks under the specification τ_x. For the other top level bindings, *i.e.* those without user-provided specifications, as well as all local bindings, LIQUIDHASKELL uses the Liquid Types [32] framework to infer refinement types, thus greatly reducing the number of annotations required from the user.

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 [40] for details.

Module	LOC	Fun	Rec	Div	Hint	Time
`GHC.List`	309	66	34	5	0	14
`Data.List`	504	97	50	2	6	11
`Data.Map.Base`	1396	180	94	0	12	175
`Data.Set.Splay`	149	35	17	0	7	26
`Bytestring`	3505	569	154	8	73	285
`Vector-Algorithms`	1218	99	31	0	31	85
`Text`	3128	493	124	5	44	481
Total	10209	1539	504	20	173	1080

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.

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 [32]. 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 10,209 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.

- *Suitable:* Our approach of eagerly proving termination is in fact, *highly* suitable: of the 504 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 65.7% of recursive functions terminating. Thus, only 34.3% 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 [29], 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 [21] 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.

Haskell itself can be used to *fake* "lightweight" dependent types [11, 16, 30]. In this style, the invariants are expressed in a restricted [22] total index language and relationships (*e.g.* $x < y$ and $y < z$) are combined (*e.g.* $x < z$) by explicitly constructing a term denoting the consequent from terms denoting the antecedents. On the plus side this "constructive" approach ensures soundness. It is impossible to witness inconsistencies, as doing so triggers diverging computations. However, it is not easy to use restricted indices with explicitly constructed relations to verify complex properties [26].

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, 33, 43]. 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 [12] which uses $\bar{\tau}$ to encode diverging terms, and types `fix` as $(\bar{\tau} \to \bar{\tau}) \to \bar{\tau}$). 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" [23, 34], or by annotating types with size indices and verifying that the arguments of recursive calls have smaller indices [3, 20]. Our use of refinements to encode terminating fixpoints is most closely related to [42], but this work also crucially assumes CBV semantics for soundness.

AProVE [18] 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 [17] 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. [44] 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 [35] 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 [41] 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. Halo's translation of Haskell programs directly encodes constructors as uninterpreted functions, axiomatized to be injective (as the denotational semantics requires). This heavyweight encoding is more precise than predicate abstraction but leads to model-theoretic problems (outlined in the Halo paper) and affects the efficiency of the encoding when scaling to larger programs (see also 8, paragraph B) in the lack of specialized decisions procedures. 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 [32] 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 proof. 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 from `good` in §2. With explicit value predicates, this subtyping reduces to the VC:

$$\begin{array}{l}(\mathsf{Val}(x) \Rightarrow x \geq 0)\\(\mathsf{Val}(y) \Rightarrow y \geq 0)\end{array} \Rightarrow (v = y + 1) \Rightarrow (v > 0) \qquad (12)$$

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 (12) 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.

Acknowledgements

We thank Kenneth Knowles, Kenneth L. McMillan, Andrey Rybalchenko, Philip Wadler, and the reviewers for their excellent suggestions and feedback about earlier versions of this paper.

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

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

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

[14] D. Detlefs, G. Nelson, and J. 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] R. A. Eisenberg and S. Weirich. Dependently typed programming with singletons. In *Haskell Symposium*, 2012.

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

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

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

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

[21] 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.

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

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

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

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

[26] S. Lindley and C. McBride. Hasochism: The pleasure and pain of dependently typed haskell programming. In *Haskell Symposium*, 2013.

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

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

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

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

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

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

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

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

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

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

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

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

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

[40] N. Vazou, E. L. Seidel, and R. Jhala. Liquidhaskell: Experience with refinement types in the real world. In *Haskell Symposium*, 2014.

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

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

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

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