LiquidHaskell: Experience with Refinement Types in the Real World *

Niki Vazou Eric L. Seidel Ranjit Jhala
UC San Diego

Abstract

Haskell has many delightful features. Perhaps the one most beloved by its users is its type system that allows developers to specify and verify a variety of program properties at compile time. However, many properties, typically those that depend on relationships *between* program values are impossible, or at the very least, cumbersome to encode within the existing type system. Many such properties can be verified using a combination of Refinement Types and external SMT solvers. We describe the refinement type checker LIQUIDHASKELL, which we have used to specify and verify a variety of properties of over 10,000 lines of Haskell code from various popular libraries, including `containers`, `hscolour`, `bytestring`, `text`, `vector-algorithms` and `xmonad`. First, we present a high-level overview of LIQUIDHASKELL, through a tour of its features. Second, we present a qualitative discussion of the kinds of properties that can be checked – ranging from generic application independent criteria like totality and termination, to application specific concerns like memory safety and data structure correctness invariants. Finally, we present a quantitative evaluation of the approach, with a view towards measuring the efficiency and programmer effort required for verification, and discuss the limitations of the approach.

1. Introduction

Refinement types enable specification of complex invariants by extending the base type system with *refinement predicates* drawn from decidable logics. For example,

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

are refinements of the basic type `Int` with a logical predicate that states the *values* `v` being described must be *non-negative* and *postive* respectively. We can specify *contracts* of functions by refining function types. For example, the contract for `div`

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

states that `div` *requires* a non-negative dividend `n` and a positive divisor `d`, and *ensures* that the result is less than the dividend. If a program (refinement) type checks, we can be sure that `div` will never throw a divide-by-zero exception.

What are refinement types good for? While there are several papers describing the *theory* behind refinement types [4, 13, 27, 29, 36, 42, 44], even for LIQUIDHASKELL [39], there is rather less literature on how the approach can be *applied* to large, real-world codes. In particular, we try to answer the following questions:

1. What properties can be specified with refinement types?
2. What inputs are provided and what feedback is received?
3. What is the process for modularly verifying a library?
4. What are the limitations of refinement types?

In this paper, we attempt to investigate these questions, by using the refinement type checker LIQUIDHASKELL, to specify and verify a variety of properties of over 10,000 lines of Haskell code from various popular libraries, including `containers`, `hscolor`, `bytestring`, `text`, `vector-algorithms` and `xmonad`. First (§ 2), we present a high-level overview of LIQUIDHASKELL, through a tour of its features. Second, we present a qualitative discussion of the kinds of properties that can be checked – ranging from generic application independent criteria like totality (§ 3), *i.e.* that a function is defined for all inputs (of a given type), and termination, (§ 4) *i.e.* that a recursive function cannot diverge, to application specific concerns like memory safety (§ 5) and functional correctness properties (§ 6). Finally (§ 7), we present a quantitative evaluation of the approach, with a view towards measuring the efficiency and programmer's effort required for verification, and we discuss various limitations of the approach which could provide avenues for further work.

2. LIQUIDHASKELL

We will start with a short description of the LIQUIDHASKELL workflow, summarized in Figure 1, and continue with an example driven overview of how properties are specified and verified using the tool.

Source LIQUIDHASKELL can be run from the command-line[1] or within a web-browser[2]. It takes as *input*: (1) a single Haskell *source* file with code and refinement type specifications including refined datatype definitions, measures (§ 2.3), predicate and type aliases, and function signatures; (2) a set of directories containing *imported modules* (including the `Prelude`) which may themselves contain specifications for exported types and functions; and (3) a set of predicate fragments called *qualifiers*, which are used to infer refine-

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

Haskell '14, September 6, 2014, Gothenburg, Sweden.
Copyright © 2014 ACM 978-1-4503-3041-1/14/09...$15.00.
http://dx.doi.org/10.1145/2633357.2633366

[1] `https://hackage.haskell.org/package/liquidhaskell`

[2] `http://goto.ucsd.edu/liquid/haskell/demo/`

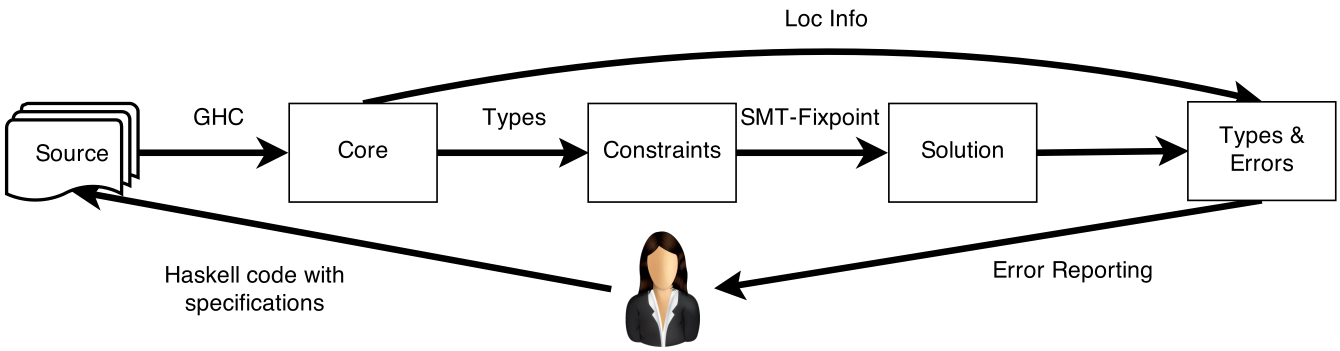

Figure 1. LIQUIDHASKELL Workflow

ment types. This set is typically empty as the default set of qualifiers extracted from the type specifications suffices for inference.

Core LIQUIDHASKELL uses GHC to reduce the source to the Core IL [35], and, to facilitate source-level error reporting, creates a map from Core expressions to locations in the Haskell source.

Constraints Then, it uses the abstract interpretation framework of Liquid Typing [29], modified to ensure soundness under lazy evaluation [39], to generate logical constraints from the Core IL.

Solution Next, it uses a fixpoint algorithm (from [29]) combined with an SMT solver to solve the constraints, and hence infers a valid refinement typing for the program. LIQUIDHASKELL can use any solver that implements the SMT-LIB2 standard [2], including Z3 [10], CVC4 [1], and MathSat [6].

Types & Errors If the set of constraints is satisfiable, then LIQUIDHASKELL outputs SAFE, meaning the program is verified. If instead, the set of constraints is not satisfiable, then LIQUIDHASKELL outputs UNSAFE, and uses the invalid constraints to report refinement type errors at the *source positions* that created the invalid constraints, using the location information to map the invalid constraints to source positions. In either case, LIQUIDHASKELL produces as output a source map containing the *inferred* types for each program expression, which, in our experience, is crucial for debugging the code and the specifications.

LIQUIDHASKELL is best thought of as an *optional* type checker for Haskell. By optional we mean that the refinements have *no* influence on the dynamic semantics, which makes it easy to apply LIQUIDHASKELL to *existing* libraries. To emphasize the optional nature of refinements and preserve compatibility with existing compilers, all specifications appear within comments of the form `{-@ ... @-}`, which we omit below for brevity.

2.1 Specifications

A refinement type is a Haskell type where each component of the type is decorated with a predicate from a (decidable) refinement logic. We use the quantifier-free logic of equality, uninterpreted functions and linear arithmetic (QF-EUFLIA) [25]. For example,

```
{v:Int | 0 <= v && v < 100}
```

describes `Int` values between `0` and `100`.

Type Aliases For brevity and readability, it is often convenient to define abbreviations for particular refinement predicates and types. For example, we can define an alias for the above predicate

```
predicate Btwn Lo N Hi = Lo <= N && N < Hi
```

and use it to define a *type alias*

```
type Rng Lo Hi = {v:Int | (Btwn Lo v Hi)}
```

We can now describe the above integers as `(Rng 0 100)`.

Contracts To describe the desired properties of a function, we need simply refine the input and output types with predicates that respectively capture suitable pre- and post-conditions. For example,

```
range :: lo:Int -> hi:{Int | lo <= hi}
      -> [(Rng lo hi)]
```

states that `range` is a function that takes two `Int`s respectively named `lo` and `hi` and returns a list of `Int`s between `lo` and `hi`. There are three things worth noting. First, we have binders to name the function's *inputs* (*e.g.*, `lo` and `hi`) and can use the binders inside the function's *output*. Second, the refinement in the *input* type describes the *pre-condition* that the second parameter `hi` cannot be smaller than the first `lo`. Third, the refinement in the *output* type describes the *post-condition* that all returned elements are between the bounds of `lo` and `hi`.

2.2 Verification

Next, consider the following implementation for `range`:

```
range lo hi
  | lo <= hi  = lo : range (lo + 1) hi
  | otherwise = []
```

When we run LIQUIDHASKELL on the above code, it reports an error at the definition of `range`. This is unpleasant! One way to debug the error is to determine what type has been *inferred* for `range`, *e.g.*, by hovering the mouse over the identifier in the web interface. In this case, we see that the output type is essentially:

```
[{v:Int | lo <= v && v <= hi}]
```

which indicates the problem. There is an *off-by-one* error due to the problematic guard. If we replace the second `<=` with a `<` and re-run the checker, the function is verified.

Holes It is often cumbersome to specify the Haskell types, as those can be gleaned from the regular type signatures or via GHC's inference. Thus, LIQUIDHASKELL allows the user to leave holes in the specifications. Suppose `rangeFind` has type

```
(Int -> Bool) -> Int -> Int -> Maybe Int
```

where the second and third parameters define a range. We can give `rangeFind` a refined specification:

```
_ -> lo:_ -> hi:{Int | lo <= hi}
  -> Maybe (Rng lo hi)
```

where the _ is simply the unrefined Haskell type for the corresponding position in the type.

Inference Next, consider the implementation

```
rangeFind f lo hi = find f $ range lo hi
```

where `find` from `Data.List` has the (unrefined) type

```
find :: (a -> Bool) -> [a] -> Maybe a
```

LIQUIDHASKELL uses the abstract interpretation framework of Liquid Typing [29] to infer that the type parameter `a` of `find` can be instantiated with `(Rng lo hi)` thereby enabling the automatic verification of `rangeFind`.

Inference is crucial for automatically synthesizing types for polymorphic instantiation sites – note there is another instantiation required at the use of the apply operator `$` – and to relieve the programmer of the tedium of specifying signatures for all functions. Of course, for functions exported by the module, we must write signatures to specify preconditions – otherwise, the system defaults to using the trivial (unrefined) Haskell type as the signature *i.e.*, checks the implementation assuming arbitrary inputs.

2.3 Measures

So far, the specifications have been limited to comparisons and arithmetic operations on primitive values. We use *measure functions*, or just measures, to specify *inductive properties* of algebraic data types. For example, we define a measure `len` to write properties about the number of elements in a list.

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

Measure definitions are *not* arbitrary Haskell code but a very restricted subset [39]. Each measure has a single equation per constructor that defines the value of the measure for that constructor. The right-hand side of the equation is a term in the restricted refinement logic. Measures are interpreted by generating refinement types for the corresponding data constructors. For example, from the above, LIQUIDHASKELL derives the following types for the list data constructors:

```
[]  :: {v:[a]| len v = 0}
(:) :: _ -> xs:_ -> {v:[a]| len v = 1 + len xs}
```

Here, `len` is an *uninterpreted function* in the refinement logic. We can define multiple measures for a type; LIQUIDHASKELL simply conjoins the individual refinements arising from each measure to obtain a single refined signature for each data constructor.

Using Measures We use measures to write specifications about algebraic types. For example, we can specify and verify that:

```
append :: xs:[a] -> ys:[a]
       -> {v:[a]| len v = len xs + len ys}

map    :: (a -> b) -> xs:[a]
       -> {v:[b]| len v = len xs}

filter :: (a -> Bool) -> xs:[a]
       -> {v:[a]| len v <= len xs}
```

Propositions Measures can be used to encode sophisticated invariants about algebraic data types. To this end, the user can write a measure whose output has a special type `Prop` denoting propositions in the refinement logic. For instance, we can describe a list that contains a `0` as:

```
measure hasZero :: [Int] -> Prop
hasZero []      = false
hasZero (x:xs)  = x == 0 || (hasZero xs)
```

We can then define lists containing a `0` as:

```
type HasZero = {v : [Int] | (hasZero v)}
```

Using the above, LIQUIDHASKELL will accept

```
xs0 :: HasZero
xs0 = [2,1,0,-1,-2]
```

but will reject

```
xs' :: HasZero
xs' = [3,2,1]
```

2.4 Refined Data Types

Often, we require that *every* instance of a type satisfies some invariants. For example, consider a `CSV` data type, that represents tables:

```
data CSV a = CSV { cols :: [String]
                 , rows :: [[a]]    }
```

With LIQUIDHASKELL we can enforce the invariant that every row in a `CSV` table should have the same number of columns as there are in the header

```
data CSV a = CSV { cols :: [String]
                 , rows :: [ListL a cols] }
```

using the alias

```
type ListL a X = {v:[a]| len v = len X}
```

A refined data definition is *global* in that LIQUIDHASKELL will reject any `CSV`-typed expression that does not respect the refined definition. For example, both of the below

```
goodCSV = CSV [  "Month", "Days"]
              [ ["Jan"  , "31"]
              , ["Feb   , "28"]
              , ["Mar"  , "31"] ]

badCSV  = CSV [  "Month", "Days"]
              [ ["Jan"  , "31"]
              , ["Feb   , "28"]
              , ["Mar"        ] ]
```

are well-typed Haskell, but the latter is rejected by LIQUIDHASKELL. Like measures, the global invariants are enforced by refining the constructors' types.

2.5 Refined Type Classes

Next, let us see how LIQUIDHASKELL supports the verification of programs that use ad-hoc polymorphism via type classes. While the implementation of each typeclass instance is different, there is often a common interface that we expect all instances to satisfy.

Class Measures As an example, consider the class definition

```
class Indexable f where
  size :: f a -> Int
  at   :: f a -> Int -> a
```

For safe access, we might require that `at`'s second parameter is bounded by the `size` of the container. To this end, we define a *type-indexed* measure, using the **`class measure`** keyword

```
class measure sz :: a -> Nat
```

Now, we can specify the safe-access precondition independent of the particular instances of `Indexable`:

```
class Indexable f where
  size :: xs:_ -> {v:Nat | v = sz xs}
  at   :: xs:_ -> {v:Nat | v < sz xs} -> a
```

Instance Measures For each concrete type that instantiates a class, we require a corresponding definition for the measure. For example, to define lists as an instance of `Indexable`, we require the definition of the `sz` instance for lists:

```
instance measure sz :: [a] -> Nat
sz []     = 0
sz (x:xs) = 1 + (sz xs)
```

Class measures work just like regular measures in that the above definition is used to refine the types of the list data constructors. After defining the measure, we can define the type instance as:

```
instance Indexable [] where
  size []        = 0
  size (x:xs)    = 1 + size xs

  (x:xs) `at` 0  = x
  (x:xs) `at` i  = index xs (i-1)
```

LIQUIDHASKELL uses the definition of `sz` for lists to check that `size` and `at` satisfy the refined class specifications.

Client Verification At the clients of a type-class we use the refined types of class methods. Consider a client of `Indexable`s:

```
sum :: (Indexable f) => f Int -> Int
sum xs = go 0
  where
    go i | i < size xs = xs `at` i + go (i+1)
         | otherwise   = 0
```

LIQUIDHASKELL proves that each call to `at` is safe, by using the refined class specifications of `Indexable`. Specifically, each call to `at` is guarded by a check `i < size xs` and `i` is increasing from 0, so LIQUIDHASKELL proves that `` xs `at` i `` will always be safe.

2.6 Abstracting Refinements

So far, all the specifications have used *concrete* refinements. Often it is useful to be able to *abstract* the refinements that appear in a specification. For example, consider a monomorphic variant of `max`

```
max     :: Int -> Int -> Int
max x y = if x > y then x else y
```

We would like to give `max` a specification that lets us verify:

```
xPos  :: {v: _ | v > 0}
xPos  = max 10 13

xNeg  :: {v: _ | v < 0}
xNeg  = max (-5) (-8)

xEven :: {v: _ | v mod 2 == 0}
xEven = max 4 (-6)
```

To this end, LIQUIDHASKELL allows the user to *abstract refinements* over types [38], for example by typing `max` as:

```
max :: forall <p :: Int -> Prop>.
         Int<p> -> Int<p> -> Int<p>
```

The above signature states that for any refinement `p`, if the two inputs of `max` satisfy `p` then so does the output. LIQUIDHASKELL uses Liquid Typing to automatically instantiate `p` with suitable concrete refinements, thereby checking `xPos`, `xNeg`, and `xEven`.

Dependent Composition Abstract refinements turn out to be a surprisingly expressive and useful specification mechanism. For example, consider the function composition operator:

```
(.) :: (b -> c) -> (a -> b) -> a -> c
(.) f g x = f (g x)
```

Previously, it was not possible to check, *e.g.* that:

```
plus3 :: x:_ -> {v:_ | v = x + 3}
plus3 = (+ 1) . (+ 2)
```

as the above required tracking the dependency between `a`, `b` and `c`, which is crucial for analyzing idiomatic Haskell. With abstract refinements, we can give the `(.)` operator the type:

```
(.) :: forall < p :: b -> c -> Prop
              , q :: a -> b -> Prop>.
          f:(x:b -> c<p x>)
       -> g:(x:a -> b<q x>)
       -> y:a
       -> exists[z:b<q y>].c<p z>
```

which gets automatically instantiated at usage sites, allowing LIQUIDHASKELL to precisely track invariants through the use of the ubiquitous higher-order operator.

Dependent Pairs Similarly, we can abstract refinements over the definition of datatypes. For example, we can express dependent pairs in LIQUIDHASKELL by refining the definition of tuples as:

```
data Pair a b <p :: a -> b -> Prop>
  = Pair { fst :: a, snd :: b<p fst>}
```

That is, the refinement `p` relates the `snd` element with the `fst`. Now we can define increasing and decreasing pairs

```
type IncP = Pair <{\x y -> x < y}> Int Int
type DecP = Pair <{\x y -> x > y}> Int Int
```

and then verify that:

```
up :: IncP
up = Pair 2 5

dn :: DecP
dn = Pair 5 2
```

Now that we have a bird's eye view of the various specification mechanisms supported by LIQUIDHASKELL, let us see how we can profitably apply them to statically check a variety of correctness properties in real-world codes.

3. Totality

Well typed Haskell code can go very wrong:

```
*** Exception: Prelude.head: empty list
```

As our first application, let us see how to use LIQUIDHASKELL to statically guarantee the absence of such exceptions, *i.e.*, to prove various functions *total*.

3.1 Specifying Totality

First, let us see how to specify the notion of totality inside LIQUIDHASKELL. Consider the source of the above exception:

```
head :: [a] -> a
head (x:_) = x
```

Most of the work towards totality checking is done by the translation to GHC's Core, in which every function *is* total, but may explicitly call an *error* function that takes as input a string that describes the source of the pattern-match failure and throws an exception. For example `head` is translated into

```
head d = case d of
           x:xs -> x
           []   -> patError "head"
```

Since every core function is total, but may explicitly call error functions, to prove that the source function is total, it suffices to prove that `patError` will *never* be called. We can specify this requirement by giving the error functions a `false` pre-condition:

```
patError :: {v:String | false } -> a
```

The pre-condition states that the input type is *uninhabited* and so an expression containing a call to `patError` will only type check if the call is *dead code*.

3.2 Verifying Totality

The (core) definition of `head` does not typecheck as is; but requires a pre-condition that states that the function is only called with non-empty lists. Formally, we do so by defining the alias

```
predicate NonEmp X = 0 < len X
```

and then stipulating that

```
head :: {v : [a] | NonEmp v} -> a
```

To verify the (core) definition of `head`, LIQUIDHASKELL uses the above signature to check the body in an environment

```
d :: {0 < len d}
```

When `d` is matched with `[]`, the environment is strengthened with the corresponding refinement from the definition of `len`, *i.e.*,

```
d :: {0 < (len d) && (len d) = 0}
```

Since the formula above is a contradiction, LIQUIDHASKELL concludes that the call to `patError` is dead code, and thereby verifies the totality of `head`. Of course, now we have pushed the burden of proof onto clients of `head` – at each such site, LIQUIDHASKELL will check that the argument passed in is indeed a `NonEmp` list, and if it successfully does so, then we, at any uses of `head`, can rest assured that `head` will never throw an exception.

Refinements and Totality While the `head` example is quite simple, in general, refinements make it very easy to prove totality in complex situations, where we must track dependencies between inputs and outputs. For example, consider the `risers` function from [24]:

```
risers []       = []
risers [x]      = [[x]]
risers (x:y:zs)
  | x <= y      = (x:s) : ss
  | otherwise   = [x] : (s:ss)
  where
    s:ss    = risers (y:etc)
```

The pattern match on the last line is partial; its core translation is

```
let (s, ss) = case risers (y:etc) of
                s:ss -> (s, ss)
                []   -> patError "..."
```

What if `risers` returns an empty list? Indeed, `risers` *does*, on occasion, return an empty list per its first equation. However, on close inspection, it turns out that *if* the input is non-empty, *then* the output is also non-empty. Happily, we can specify this as:

```
risers :: l:_ -> {v:_ | NonEmp l => NonEmp v}
```

LIQUIDHASKELL verifies that `risers` meets the above specification, and hence that the `patError` is dead code as at that site, the scrutinee is obtained from calling `risers` with a `NonEmp` list.

Non-Emptiness via Measures Instead of describing non-emptiness indirectly using `len`, a user could a special measure:

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

predicate NonEmp X = nonEmp X
```

After which, verification would proceed analagous to the above.

Total Totality Checking `patError` is one of many possible errors thrown by non-total functions. `Control.Exception.Base` has several others (`recSelError`, `irrefutPatError`, *etc.*) which serve the purpose of making core translations total. Rather than hunt down and specify `false` preconditions one by one, the user may automatically turn on totality checking by invoking LIQUIDHASKELL with the `--totality` command line option, at which point the tool systematically checks that all the above functions are indeed dead code, and hence, that all definitions are total.

3.3 Case Studies

We verified totality of two libraries: `HsColour` and `Data.Map`, earlier versions of which had previously been proven total by `catch` [24].

`Data.Map` is a widely used library for (immutable) key-value maps, implemented as balanced binary search trees. Totality verification of `Data.Map` was quite straightforward. We had previously verified termination and the crucial binary search invariant [38]. To verify totality it sufficed to simply re-run verification with the `--totality` argument. All the important specifications were already captured by the types, and no additional changes were needed to prove totality.

This case study illustrates an advantage of LIQUIDHASKELL over specialized provers (*e.g.*, `catch` [24]), namely it can be used to prove totality, termination and functional correctness at the same time, facilitating a nice reuse of specifications for multiple tasks.

`HsColour` is a library for generating syntax-highlighted LATEX and HTML from Haskell source files. Checking `HsColour` was not so easy, as in some cases assumptions are used about the structure of the input data: For example, `ACSS.splitSrcAndAnnos` handles an input list of `String`s and assumes that whenever a specific `String` (say `breakS`) appears then at least two `String`s (call them `mname` and `annots`) follow it in the list. Thus, for a list `ls` that starts with `breakS` the irrefutable pattern `(_:mname:annots)= ls` should be total. Currently it is somewhat cumbersome to specify such properties, and these are interesting avenues for future work. Thus to prove totality, we added a dynamic check that validates that the length of the input `ls` exceeds `2`.

In other cases assertions were imposed via monadic checks, for example `HsColour.hs` reads the input arguments and checks their well-formedness using

```
when (length f > 1) $ errorOut "..."
```

Currently LIQUIDHASKELL does not support monadic reasoning that allows assuming that `(length f <= 1)` holds when executing the action *following* the `when` check. Finally, code modifications were required to capture properties that currently we do not know how to express with LIQUIDHASKELL. For example, `trimContext` checks if there is an element that satisfies `p` in the list `xs`; if so it defines `ys = dropWhile (not . p)xs` and computes `tail ys`. By the check we know that `ys` has at least one element, the one that satisfies `p`, but this is a property that we could not express in LIQUIDHASKELL.

On the whole, while proving totality can be cumbersome (as in `HsColour`) it is a nice side benefit of refinement type checking, and can sometimes be a fully automatic corollary of establishing more interesting safety properties (as in `Data.Map`).

4. Termination

To soundly account for Haskell's non-strict evaluation, a refinement type checker must distinguish between terms that may potentially diverge and those that will not [39]. Thus, by default, LIQUIDHASKELL proves termination of each recursive function. Fortunately, refinements make this onerous task quite straightforward. We need simply associate a *well-founded termination metric* on the function's parameters, and then use refinement typing to check that the metric strictly decreases at each recursive call. In practice, due to a careful choice of defaults, this amounts to about a line of termination-related hints per hundred lines of source. Details about the termination checker may be found in [39], we include a brief description here to make the paper self-contained.

Simple Metrics As a starting example, consider the `fac` function

```
fac :: n:Nat -> Nat / [n]
```

```
fac 0 = 1
fac n = n * fac (n-1)
```

The termination metric is simply the parameter `n`; as `n` is non-negative and decreases at the recursive call, LIQUIDHASKELL verifies that `fac` will terminate. We specify the termination metric in the type signature with the `/[n]`.

Termination checking is performed at the same time as regular type checking, as it can be reduced to refinement type checking with a special terminating fixpoint combinator [39]. Thus, if LIQUIDHASKELL fails to prove that a given termination metric is well-formed and decreasing, it will report a `Termination Check Error`. At this point, the user can either debug the specification, or mark the function as non-terminating.

Termination Expressions Sometimes, no single parameter decreases across recursive calls, but there is some *expression* that forms the decreasing metric. For example recall `range lo hi` (from § 2.2) which returns the list of `Int`s from `lo` to `hi`:

```
range lo hi
  | lo < hi   = lo : range (lo+1) hi
  | otherwise = []
```

Here, neither parameter is decreasing (indeed, the first one is increasing) but `hi-lo` decreases across each call. To account for such cases, we can specify as the termination metric a (refinement logic) expression over the function parameters. Thus, to prove termination, we could type `range` as:

```
lo:Int -> hi:Int -> [(Btwn lo hi)] / [hi-lo]
```

Lexicographic Termination The Ackermann function

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

is curious as there exists no simple, natural-valued, termination metric that decreases at each recursive call. However `ack` terminates because at each call *either* `m` decreases *or* `m` remains the same and `n` decreases. In other words, the pair `(m,n)` strictly decreases according to a *lexicographic* ordering. Thus LIQUIDHASKELL supports termination metrics that are a *sequence of* termination expressions. For example, we can type `ack` as:

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

At each recursive call LIQUIDHASKELL uses a lexicographic ordering to check that the sequence of termination expressions is decreasing (and well-founded in each component).

Mutual Recursion The lexicographic mechanism lets us check termination of mutually recursive functions, *e.g.* `isEven` and `isOdd`

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

isOdd n  = not $ isEven n
```

Each call terminates as either `isEven` calls `isOdd` with a decreasing parameter, *or* `isOdd` calls `isEven` with the same parameter, expecting the latter to do the decreasing. For termination, we type:

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

To check termination, LIQUIDHASKELL verifies that at each recursive call the metric of the caller is less than the metric of the callee. When `isEven` calls `isOdd`, it proves that the caller's metric, namely `[n,0]` is greater than the callee's `[n-1,1]`. When `isOdd` calls `isEven`, it proves that the caller's metric `[n,1]` is greater than the callee's `[n,0]`, thereby proving the mutual recursion always terminates.

Recursion over Data Types The above strategies generalize 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 example, we can prove that `map`

```
map f (x:xs) = f x : map f xs
map f []     = []
```

terminates, by typing `map` as

```
(a -> b) -> xs:[a] -> [b] / [len xs]
```

i.e., by using the measure `len xs`, from § 2.3, as the metric.

Generalized Metrics Over Datatypes In many functions there is no single argument whose measure provably decreases. Consider

```
merge (x:xs) (y:ys)
  | x < y     = x : merge xs (y:ys)
  | otherwise = y : merge (x:xs) ys
```

from the homonymous sorting routine. Here, neither parameter decreases, but the *sum* of their sizes does. To prove termination, we can type `merge` as:

```
xs:[a] -> ys:[a] -> [a] / [len xs + len ys]
```

Putting it all Together The above techniques can be combined to prove termination of the mutually recursive quick-sort (from [41])

```
qsort (x:xs)   = qpart x xs [] []
qsort []       = []

qpart x (y:ys) l r
  | x > y      = qpart x ys (y:l) r
  | otherwise  = qpart x ys l (y:r)
qpart x [] l r = app x (qsort l) (qsort r)

app k []     z = k : z
app k (x:xs) z = x : app k xs z
```

`qsort (x:xs)` calls `qpart x xs` to partition `xs` into two lists `l` and `r` that have elements less and greater or equal than the pivot `x`, respectively. When `qpart` finishes partitioning it mutually recursively calls `qsort` to sort the two list and appends the results with `app`. LIQUIDHASKELL proves sortedness as well [38] but let us focus here on termination. To this end, we type the functions as:

```
qsort :: xs:_ -> _
      / [len xs, 0]

qpart :: _ -> ys:_ -> l:_ -> r:_ -> _
      / [len ys + len l + len r, 1 + len ys]
```

As before, LIQUIDHASKELL checks that at each recursive call the caller's metric is less than the callee's. When `qsort` calls `qpart` the length of the unsorted list `len (x:xs)` exceeds the `len xs + len [] + len []`. When `qpart` recursively calls itself the first component of the metric is the same, but the length of the unpartitioned list decreases, *i.e.* `1 + len y:ys` exceeds `1 + len ys`. Finally, when `qpart` calls `qsort` we have `len ys + len l + len r` exceeds both `len l` and `len r`, thereby ensuring termination.

Automation: Default Size Measures The `qsort` example illustrates that while LIQUIDHASKELL is very expressive, devising appropriate termination metrics can be tricky. Fortunately, such patterns are very uncommon, and the vast majority of cases in real world programs are just structural recursion on a datatype. LIQUIDHASKELL automates termination proofs for this common case, by allowing users to specify a *default size measure* for each data type, *e.g.* `len` for `[a]`. Now, if no explicit 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 `map` as the size measure is automatically used to prove termination. This heuristic suffices to *automatically* prove 67% of recursive functions terminating.

Disabling Termination Checking In `Haskell`'s lazy setting not all functions are terminating. LIQUIDHASKELL provides two mechanisms the disable termination proving. A user can disable checking a single function by marking that function as lazy. For example, specifying **`lazy`** `repeat` tells the tool to not prove `repeat` terminates. Optionally, a user can disable termination checking for a whole module by using the command line argument `--no-termination` for the entire file.

5. Memory Safety

The terms "Haskell" and "pointer arithmetic" rarely occur in the same sentence, yet many Haskell programs are constantly manipulating pointers under the hood by way of using the `Bytestring` and `Text` libraries. These libraries sacrifice safety for (much needed) speed and are therefore natural candidates for verification through LIQUIDHASKELL.

5.1 Bytestring

The single most important aspect of the `Bytestring` library,our first case study, is its pervasive intermingling of high level abstractions like higher-order loops, folds, and fusion, with low-level pointer manipulations in order to achieve high-performance. `Bytestring` is an appealing target for evaluating LIQUIDHASKELL, as refinement types are an ideal way to statically ensure the correctness of the delicate pointer manipulations, errors in which lie below the scope of dynamic protection.

The library spans 8 files (modules) totaling about 3,500 lines. We used LIQUIDHASKELL to verify the library by giving precise types describing the sizes of internal pointers and bytestrings. These types are used in a modular fashion to verify the implementation of functional correctness properties of higher-level API functions which are built using lower-level internal operations. Next, we show the key invariants and how LIQUIDHASKELL reasons precisely about pointer arithmetic and higher-order codes.

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

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

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

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

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

```
type ByteStringEq B
  = {v:ByteString | (bLen v) = (bLen B)}
copy :: b:ByteString -> ByteStringEq b
copy (PS fp off len)
  = unsafeCreate len $ \p ->
      withForeignPtr fp $ \f ->
        memcpy len p (f `plusPtr` off)
```

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

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

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

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

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

Interfacing with the Real World The above illustrates how LIQUIDHASKELL analyzes code that interfaces with the "real world" via the C FFI. We specify the behavior of the world via a refinement typed interface. These types are then assumed to hold for the corresponding functions, *i.e.* generate pre-condition checks and post-condition guarantees at usage sites within the Haskell code.

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

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

To enable fusion [9] `loopDown` uses a higher order `loopWrapper` to iterate over the buffer with a `doDownLoop` action:

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

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

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

The function `doDownLoop` is an internal function. Via abstract interpretation [29], LIQUIDHASKELL infers that (1) `len` is less than the sizes of `src` and `dest`, (2) `f` (here, `mapAccumEFL`) always returns a `JustS`, so (3) source and destination offsets satisfy $0 \leq \mathtt{s}, \mathtt{d} < \mathtt{len}$, (4) the generated `IO` action returns a triple `(acc :*: 0 :*: len)`, thereby proving the safety of the accesses in `loop` *and* verifying that `loopDown` and the API function `mapAccumR` return a `Bytestring` whose size equals its input's.

To prove *termination*, we add a *termination expression* `s+1` which is always non-negative and decreases at each call.

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

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

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

LIQUIDHASKELL uses the above to verify that

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

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

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

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

Via inference, LIQUIDHASKELL verifies the safety of the pointer accesses, and determines that the sum of the lengths of the output pair of `ByteString`s equals that of the input `ps`. `go` terminates as `l-i` is a well-founded decreasing metric.

5.2 Text

Next we present a brief overview of the verification of `Text`, which is the standard library used for serious unicode text processing. `Text` uses byte arrays and stream fusion to guarantee performance while providing a high-level API. In our evaluation of LIQUIDHASKELL on `Text`,we focused on two types of properties: (1) the safety of array index and write operations, and (2) the functional correctness of the top-level API. These are both made more interesting by the fact that `Text` internally encodes characters using UTF-16, in which characters are stored in either two or four bytes. `Text` is a vast library spanning 39 modules and 5,700 lines of code, however we focus on the 17 modules that are relevant to the above properties. While we have verified exact functional correctness size properties for the top-level API, we focus here on the low-level functions and interaction with unicode.

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

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

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

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

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

The UTF-16 encoding complicates the specification of the function as we cannot simply require `i` to be less than the length of `marr`; if `i` were `malen marr - 1` and `c` required two `Word16`s, we would perform an out-of-bounds write. We account for this subtlety with a predicate that states there is enough `Room` to encode `c`.

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

```
                    else OkN I A 2

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

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

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

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

```
outer arr top = loop
 where
  loop !z !s !i =
    case next0 s of
      Done          -> return (arr, (z,i))
      Skip s'       -> loop z s' i
      Yield x s'
        | j >= top  -> do
          let top' = (top + 1) `shiftL` 1
          arr' <- new top'
          copyM arr' 0 arr 0 top
          outer arr' top' z s i
        | otherwise -> do
          let (z',c) = f z x
          d <- writeChar arr i c
          loop z' s' (i+d)
        where j | ord x < 0x10000 = i
                | otherwise       = i + 1
```

Let's focus on the `Yield x s'` case. We first compute the maximum index `j` to which we will write and determine the safety of a write. If it is safe to write to `j` we call the provided function `f` on the accumulator `z` and the character `x`, and write the *resulting* character `c` into the array. However, we know nothing about `c`, in particular, whether `c` will be stored as one or two `Word16`s! Thus, LIQUIDHASKELL flags the call to `writeChar` as *unsafe*. The error can be fixed by lifting `f z x` into the **`where`** clause and defining the write index `j` by comparing `ord c` (not `ord x`). LIQUIDHASKELL (and the authors) readily accepted our fix.

6. Functional Correctness Invariants

So far, we have considered a variety of general, application independent correctness criteria. Next, let us see how we can use LIQUIDHASKELL to specify and statically verify critical application specific correctness properties, using two illustrative case studies: red-black trees, and the stack-set data structure introduced in the `xmonad` system.

6.1 Red-Black Trees

Red-Black trees have several non-trivial invariants that are ideal for illustrating the effectiveness of refinement types, and contrasting with existing approaches based on GADTs [19]. The structure can be defined via the following Haskell type:

```
data Col    = R | B
data Tree a = Leaf
            | Node Col a (Tree a) (Tree a)
```

However, a `Tree` is a valid Red-Black tree only if it satisfies three crucial invariants:

- ***Order:*** The keys must be binary-search ordered, *i.e.* the key at each node must lie between the keys of the left and right subtrees of the node,
- ***Color:*** The children of every *red* `Node` must be colored *black*, where each `Leaf` can be viewed as black,
- ***Height:*** The number of black nodes along any path from each `Node` to its `Leaf`s must be the same.

Red-Black trees are especially tricky as various operations create trees that can temporarily violate the invariants. Thus, while the above invariants can be specified with singletons and GADTs, encoding all the properties (and the temporary violations) results in a proliferation of data constructors that can somewhat obfuscate correctness. In contrast, with refinements, we can specify and verify the invariants in isolation (if we wish) and can trivially compose them simply by *conjoining* the refinements.

Color Invariant To specify the color invariant, we define a *black-rooted tree* as:

```
measure isB            :: Tree a -> Prop
color (Node c x l r)  = c == B
color (Leaf)          = true
```

and then we can describe the color invariant simply as:

```
measure isRB         :: Tree a -> Prop
isRB (Leaf)          = true
isRB (Node c x l r) = isRB l && isRB r &&
                      c = R => (isB l && isB r)
```

The insertion and deletion procedures create intermediate *almost* red-black trees where the color invariant may be violated at the root. Rather than create new data constructors we can define almost red-black trees with a measure that just drops the invariant at the root:

```
measure almostRB         :: Tree a -> Prop
almostRB (Leaf)          = true
almostRB (Node c x l r) = isRB l && isRB r
```

Height Invariant To specify the height invariant, we define a black-height measure:

```
measure bh         :: Tree a -> Int
bh (Leaf)          = 0
bh (Node c x l r) = bh l
                   + if c = R then 0 else 1
```

and we can now specify black-height balance as:

```
measure isBal         :: Tree a -> Prop
isBal (Leaf)          = true
isBal (Node c x l r) = bh l = bh r
                      && isBH l && isBH r
```

Note that `bh` only considers the left sub-tree, but this is legitimate, because `isBal` will ensure the right subtree has the same `bh`.

Order Invariant Finally, to encode the binary-search ordering property, we parameterize the datatype with abstract refinements:

```
data Tree a <l::a->a->Prop, r::a->a->Prop>
  = Leaf
  | Node { c   :: Col
         , key :: a
         , lt  :: Tree<l,r> a<l key>
         , rt  :: Tree<r,r> a<r key> }
```

Intuitively, `l` and `r` are relations between the root `key` and *each* element in its left and right subtree respectively. Now the alias:

```
type OTree a
  = Tree <{\k v -> v<k}, {\k v -> v>k}> a
```

describes binary-search ordered trees!

Composing Invariants Finally, we can compose the invariants, and define a Red-Black tree with the alias:

```
type RBT a = {v:OTree a | isRB v && isBal v}
```

An almost Red-Black tree is the above with `isRB` replaced with `almostRB`, *i.e.* does not require any new types or constructors. If desired, we can ignore a particular invariant simply by replacing the corresponding refinement above with `true`. Given the above – and suitable signatures LIQUIDHASKELL verifies the various insertion, deletion and rebalancing procedures for a Red-Black Tree library.

6.2 Stack Sets in XMonad

`xmonad` is a dynamically tiling `X11` window manager that is written and configured in Haskell. The set of windows managed by XMonad is organized into a hierarchy of types. At the lowest level we have a *set* of windows `a` represented as a `Stack a`

```
data Stack a = Stack { focus :: a
                     , up    :: [a]
                     , down  :: [a] }
```

The above is a zipper [16] where `focus` is the "current" window and `up` and `down` the windows "before" and "after" it. Each `Stack` is wrapped inside a `Workspace` that has additional information about layout and naming:

```
data Workspace i l a = Workspace
   { tag    :: i
   , layout :: l
   , stack  :: Maybe (Stack a) }
```

which is in turn, wrapped inside a `Screen`:

```
data Screen i l a sid sd = Screen
  { workspace    :: Workspace i l a
  , screen       :: sid
  , screenDetail :: sd }
```

The set of all screens is represented by the top-level zipper:

```
data StackSet i l a sid sd = StackSet
  { cur :: Screen i l a sid sd
  , vis :: [Screen i l a sid sd]
  , hid :: [Workspace i l a]
  , flt :: M.Map a RationalRect }
```

Key Invariant: Uniqueness of Windows The key invariant for the `StackSet` type is that each window `a` should appear at most once in a `StackSet i l a sid sd`. That is, a window should *not be duplicated* across stacks or workspaces. Informally, we specify this invariant by defining a measure for the *set of elements* in a list, `Stack`, `Workspace` and `Screen`, and then we use that measure to assert that the relevant sets are disjoint.

Specification: Unique Lists To specify that the set of elements in a list is unique, *i.e.* there are no duplicates in the list we first define a measure denoting the set using Z3's [10] built-in theory of sets:

```
measure elts :: [a] -> Set a
elts ([])   = emp
elts (x:xs) = cup (sng x) (elts xs)
```

Now, we can use the above to define uniqueness:

```
measure isUniq :: [a] -> Prop
isUniq ([])   =  true
isUniq (x:xs) =  notIn x xs && isUniq xs
```

where `notIn` is an abbreviation:

```
predicate notIn X S = not (mem X (elts S))
```

Specification: Unique Stacks We can use `isUniq` to define unique, *i.e.*, duplicate free, `Stack`s as:

```
data Stack a = Stack
 { focus :: a
 , up    :: {v:[a] | Uniq1 v focus}
 , down  :: {v:[a] | Uniq2 v focus up} }
```

using the aliases

```
predicate Uniq1 V X
  = isUniq V  && notIn X V
predicate Uniq2 V X Y
  = Uniq1 V X && disjoint Y V
predicate disjoint X Y
  = cap (elts X) (elts Y) = emp
```

i.e. the field `up` is a unique list of elements different from `focus`, and the field `down` is additionally disjoint from `up`.

Specification: Unique StackSets It is straightforward to lift the `elts` measure to the `Stack` and the wrapper types `Workspace` and `Screen`, and then correspondingly lift `isUniq` to `[Screen]` and `[Workspace]`. Having done so, we can use those measures to refine the type of `StackSet` to stipulate that there are no duplicates:

```
type UniqStackSet i l a sid sd
  = {v: StackSet i l a sid sd | NoDups v}
```

using the predicate aliases

```
predicate NoDups V
  =  disjoint3 (hid V) (cur V) (vis V)
  && isUniq (vis V)
  && isUniq (hid V)

predicate disjoint3 X Y Z
  =  disjoint X Y
  && disjoint Y Z
  && disjoint X Z
```

LIQUIDHASKELL automatically turns the record selectors of refined data types to measures that return the values of appropriate fields, hence `hid x` (resp. `cur x`, `vis x`) are the values of the `hid`, `cur` and `vis` fields of a `StackSet` named `x`.

Verification LIQUIDHASKELL uses the above refined type to verify the key invariant, namely, that no window is duplicated. Three key actions of the, eventually successful, verification process can be summarized as follows:

- *Strengthening library functions.* `xmonad` repeatedly concatenates the lists of a Stack. To prove that for some `s:Stack a`, `(up s ++ down s)` is a unique list, the type of `(++)` needs to capture that concatenation of two unique and disjoint lists is a unique list. For verification, we assumed that Prelude's `(++)` satisfies this property. But, not all arguments of `(++)` are unique disjoint lists: `"StackSet"++"error"` is a trivial example that does not satisfy the assumed preconditions of `(++)` thus creating a type error. Currently, LIQUIDHASKELL does not support intersection types, thus we used an unrefined `(++.)` variant of `(++)` for such cases.
- *Restrict the functions' domain.* `modify` is a `maybe`-like function that, given a default value `x`, a function `f`, and a StackSet `s`, applies `f` on the `Maybe (Stack a)` values inside `s`.

```
modify :: x:{v:Maybe (Stack a) | isNothing v}
       -> (y:Stack a
          -> Maybe {v:Stack a | SubElts v y})
       -> UniqStackSet i l a s sd
       -> UniqStackSet i l a s sd
```

Since inside the StackSet `s` each `y:Stack a` could be replaced with either the default value `x` or with `f y`, we need to ensure that both these alternatives will not insert duplicates. This imposes the curious precondition that the default value should be `Nothing`.

- *Code inlining* Given a tag `i` and a StackSet `s`, `view i s` will set the current Screen to the screen with tag `i`, if such a screen exists in `s`. Below is the original definition for `view` in case when a screen with tag `i` exists in visible screens

```
view :: (Eq s, Eq i) => i
     -> StackSet i l a s sd
     -> StackSet i l a s sd
view i s
  | Just x <- find ((i==).tag.workspace)
                   (visible s)
  = s { current = x
      , visible = current s
                : deleteBy (equating screen) x
                           (visible s) }
```

Verification of this code is difficult as we cannot suitably type `find`. Instead we *inline* the call to `find` and the field update into a single recursive function `raiseIfVisible i s` that in-place replaces `x` with the current screen.

Finally, `xmonad` comes with an extensive suite of QuickCheck properties, that were formally verified in Coq [37]. In future work, it would be interesting to do a similar verification with LIQUIDHASKELL, to compare the refinement types to proof-assistants.

7. Evaluation

We now turn to a quantitative evaluation of our experiments with LIQUIDHASKELL.

7.1 Results

We have used the following libraries as benchmarks:

- `GHC.List` and `Data.List`, which together implement many standard list operations; we verify various size related properties,
- `Data.Set.Splay`, which implements a splay-tree based functional set data type; we verify that all interface functions terminate and return well ordered trees,
- `Data.Map.Base`, which implements a functional map data type; we verify that all interface functions terminate and return binary-search ordered trees [38],
- `HsColour`, a syntax highlighting program for Haskell code, we verify totality of all functions (§ 3),
- `XMonad`, a tiling window manager for X11, we verify the uniqueness invariant of the core datatype, as well as some of the QuickCheck properties (§ 6.2),
- `Bytestring`, a library for manipulating byte arrays, we verify termination, low-level memory safety, and high-level functional correctness properties (§ 5.1),
- `Text`, a library for high-performance unicode text processing; we verify various pointer safety and functional correctness properties (§ 5.2), during which we find a subtle bug,
- `Vector-Algorithms`, which includes a suite of "imperative" (*i.e.* monadic) array-based sorting algorithms; we verify the correctness of vector accessing, indexing, and slicing *etc.*.

Table 1 summarizes our experiments, which covered 56 modules totaling 11,512 non-comment lines of source code and 1,975 lines of specifications. The results are on a machine with an Intel Xeon X5660 and 32GB of RAM (no benchmark required more than 1GB.) The upshot is that LIQUIDHASKELL is very effective on real-world code bases. The total overhead due to hints, *i.e.* the sum of **Annot** and **Qualif**, is 3.5% of **LOC**. The specifications themselves are machine checkable versions of the comments placed around functions describing safe usage and behavior, and required around two lines to express on average. While there is much room for improving the running times, the tool is fast enough to be used interactively, verify a handful of API functions and associated helpers in isolation.

7.2 Limitations

Our case studies also highlighted several limitations of LIQUIDHASKELL that we will address in future work. In most cases, we could alter the code slightly to facilitate verification.

Ghost parameters are sometimes needed in order to materialize values that are not needed for the computation, but are necessary to prove various specifications. For example, the `piv` parameter in the `append` function for red-black trees (§ 6.1).

Fixed-width integer and floating-point numbers LIQUIDHASKELL uses the theories of linear arithmetic and real numbers to reason about numeric operations. In some cases this causes us to lose precision, *e.g.* when we have to approximate the behavior of bitwise operations. We could address this shortcoming by using the theory of bit-vectors to model fixed-width integers, but we are unsure of the effect this would have on LIQUIDHASKELL's performance.

Higher-order functions must sometimes be *specialized* because the original type is not precise enough. For example, the `concat` function that concatenates a list of input `ByteString`s pre-allocates the output region by computing the total size of the input.

```
len = sum . map length $ xs
```

Unfortunately, the type for `map` is not sufficiently precise to conclude that the value `len` equals `bLens xs`, se we must manually specialize the above into a single recursive traversal that computes the lengths. Rather than complicating the type system with a very general higher-order type for `map` we suspect the best way forward will be to allow the user to specify inlining in a clean fashion.

Functions as Data Several libraries like `Text` encode data structures like (finite) streams using functions, in order to facilitate fusion. Currently, it is not possible to describe sizes of these structures using measures, as this requires describing the sizes of input-output chains starting at a given seed input for the function. In future work, it will be interesting to extend the measure mechanism to support multiple parameters (*e.g.* a stream and a seed) in order to reason about such structures.

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

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

Error messages are a crucial part of any type-checker. Currently, we report error locations in the provided source file and output

Module	Version	LOC	Mod	Fun	Specs	Annot	Qualif	Time (s)
GHC.LIST	7.4.1	309	1	66	29 / 38	6 / 6	0 / 0	15
DATA.LIST	4.5.1.0	504	1	97	15 / 26	6 / 6	3 / 3	11
DATA.MAP.BASE	0.5.0.0	1396	1	180	125 / 173	13 / 13	0 / 0	174
DATA.SET.SPLAY	0.1.1	149	1	35	27 / 37	5 / 5	0 / 0	27
HSCOLOUR	1.20.0.0	1047	16	234	19 / 40	5 / 5	1 / 1	196
XMONAD.STACKSET	0.11	256	1	106	74 / 213	3 / 3	4 / 4	27
BYTESTRING	0.9.2.1	3505	8	569	307 / 465	55 / 55	47 / 124	294
TEXT	0.11.2.3	3128	17	493	305 / 717	52 / 54	49 / 97	499
VECTOR-ALGORITHMS	0.5.4.2	1218	10	99	76 / 266	9 / 9	13 / 13	89
Total		11512	56	1879	977 / 1975	154 / 156	117 / 242	1336

Table 1. A quantitative evaluation of our experiments. **Version** is version of the checked library. **LOC** is the number of non-comment lines of source code as reported by `sloccount`. **Mod** is the number of modules in the benchmark and **Fun** is the number of functions. **Specs** is the number (/ line-count) of type specifications and aliases, data declarations, and measures provided. **Annot** is the number (/ line-count) of other annotations provided, these include invariants and hints for the termination checker. **Qualif** is the number (/ line-count) of provided qualifiers. **Time (s)** is the time, in seconds, required to run LIQUIDHASKELL.

the failed constraint(s). Unfortunately, the constraints often refer to intermediate values that have been introduced during the ANF-transformation, which obscures their relation to the program source. In future work, we may attempt to map these intermediate values back to their source expressions, which should increase the comprehensibility of our error messages. Another interesting possibility would be to search for concrete counterexamples when LIQUIDHASKELL detects an invalid constraint.

8. Related Work

Next, we situate LIQUIDHASKELL with existing Haskell verifiers.

Dependent Types are the basis of many verifiers, or more generally, proof assistants. Verification of haskell code is possible with "full" dependently typed systems like Coq [5], Agda [26], Idris [7], Omega [33], and $\lambda_{\rightarrow}$ [22]. While these systems are highly expressive, their expressiveness comes at the cost of making logical validity checking undecidable thus rendering verification cumbersome. Haskell itself can be considered a dependently-typed language, as type level computation is allowed via Type Families [23], Singleton Types[12], Generalized Algebraic Datatypes (GADTs) [28, 31], and type-level functions [8]. Again, verification in haskell itself turns out to be quite painful [21].

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 [4, 11, 30, 42]. LIQUIDHASKELL uses Liquid Types [20] that restrict the invariants even more to allow type inference, a crucial feature of a usable type system. Even though the language of refinements is restricted, as we presented, the combination of Abstract Refinements [38] with sophisticated measure definitions allows specification and verification of a wide variety of program properties.

Static Contract Checkers like ESCJava [14] are a classical way of verifying correctness through assertions and pre- and post-conditions. [43] 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 [34] is an automatic Haskell prover that combines unrolling with heuristics for rewriting and proof-search. Finally, the Halo [40] 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.

Totality Checking is feasible by GHC itself, via an option flag that warns of any incomplete patterns. Regrettably, GHC's warnings are local, *i.e.* GHC will raise a warning for `head`'s partial definition, but not for its caller, as the programmer would desire. Catch [24], a fully automated tool that tracks incomplete patterns, addresses the above issue by computing functions' pre- and post-conditions. Moreover, catch statically analyses the code to track reachable incomplete patterns. LIQUIDHASKELL allows more precise analysis than catch, thus, by assigning the appropriate types to $\star$Error functions (§ 3) it tracks reachable incomplete patters as a side-effect of verification.

Termination Analysis is crucial for LIQUIDHASKELL's soundness [39] and is implemented in a technique inspired by [41], Various other authors have proposed techniques to verify termination of recursive functions, either using the "size-change principle" [18, 32], or by annotating types with size indices and verifying that the arguments of recursive calls have smaller indices [3, 17]. To our knowledge, none of the above analyses have been empirically evaluated on large and complex real-world libraries.

AProVE [15] implements a powerful, fully-automatic termination analysis for Haskell based on term-rewriting. Compared to AProVE, encoding the termination proof via refinements provides 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.* `qsort`), (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.

9. Conclusion

We presented LIQUIDHASKELL, a refinement type checker for Haskell programs. Specifically, we presented a high-level overview of LIQUIDHASKELL, through a tour of its features; a qualitative discussion of the kinds of properties that can be checked; and a quantitative evaluation of the approach.

LIQUIDHASKELL users, especially the ones coming from a dependent type theory background, should keep in mind that the refinement language is not arbitrary haskell terms. Instead it is a restricted logical language determined by the underlying SMT solver. Thus, a natural question that arises is: "What kinds of properties or constructs can(not) be verified by LIQUIDHASKELL?" Unfortunately, we have no such answers since the boundaries of what is possible are constantly expanding, either by improvements in the tool, by creatively encoding specifications [38], or by modifying the code slightly to facilitate verification. Indeed, to appreciate the difficulty of answering this question, replace LIQUIDHASKELL with "Haskell's type system!" Instead, over the course of this work we have qualitatively circumscribed the wide space of use cases for refinement types, and have identified some lacunae that may be addressed in future work. Ultimately, we hope that with more users and experience, we will be able identify various common specification patterns or idioms to easily demarcate the boundary of what is possible with automatic SMT-based verification.

Acknowledgements

We thank Dimitrios Vytiniotis and Simon Peyton-Jones for inspiring our work on checking totality. Also, we thank Colin S. Gordon and all LIQUIDHASKELL's users for their feedback and bug reports when using the tool.

References

[1] C. Barrett, C. Conway, M. Deters, L. Hadarean, D. Jovanović, T. King, A. Reynolds, and C. Tinelli. CVC4. In *CAV*, 2011.

[2] C. Barrett, A. Stump, and C. Tinelli. The SMT-LIB Standard: Version 2.0. 2010.

[3] G. Barthe, M. J. Frade, E. Giménez, L. Pinto, and T. Uustalu. Type-based termination of recursive definitions. In *MSCS*, 2004.

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

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

[6] M. Bozzano, R. Bruttomesso, A. Cimatti, T. Junttila, P. Rossum, S. Schulz, and R. Sebastiani. MathSAT: Tight integration of SAT and mathematical decision procedures. *J. Autom. Reason.*, 2005.

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

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

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

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

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

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

[13] C. Flanagan. Hybrid type checking. In *POPL*, 2006.

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

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

[16] G. P. Huet. The Zipper. *J. Funct. Program.*, 1997.

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

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

[19] S. Kahrs. Red-black trees with types. *J. Funct. Program.*, 2001.

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

[21] S. Lindley and C. McBride. Hasochism: the pleasure and pain of dependently typed Haskell programming. In *Haskell*, 2013.

[22] A. Löh, C. McBride, and W. Swierstra. A tutorial implementation of a dependently typed lambda calculus. *Fundam. Inform.*, 2010.

[23] C. McBride. Faking it: Simulating dependent types in Haskell. *J. Funct. Program.*, 2002.

[24] N. Mitchell and C. Runciman. Not all patterns, but enough - an automatic verifier for partial but sufficient pattern matching. In *Haskell*, 2008.

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

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

[27] S. Owre, J.M. Rushby, and N. Shankar. PVS: A prototype verification system. In *CADE*, 1992.

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

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

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

[31] T. Schrijvers, S. L. Peyton-Jones, M. Sulzmann, and D. Vytiniotis. Complete and decidable type inference for GADTs. In *ICFP*, 2009.

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

[33] T. Sheard. Type-level computation using narrowing in omega. In *PLPV*, 2006.

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

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

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

[37] W. Swierstra. Xmonad in Coq (experience report): Programming a window manager in a proof assistant. In *Haskell Symposium*, 2012.

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

[39] N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. Peyton-Jones. Refinement types for haskell. In *ICFP*, 2014.

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

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

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

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

[44] C. Zenger. Indexed types. *TCS*, 1997.