Tutorial: Type-Based Analysis of Higher-Order Programs

Niki Vazou[1], Patrick M. Rondon[2], Eric Seidel[1], and Ranjit Jhala[1]

[1]UC San Diego [2]Google

Abstract. We present LIQUIDHASKELL, a verifier for Haskell programs which uses Liquid Types to reduce the verification of higher-order, polymorphic, recursive programs over complex data types, into first-order *Horn Clauses* over integers and booleans which are then solved using classical predicate abstraction. In this tutorial, we will present an overview of this approach, and then describe a simple technique called *abstract refinements* which greatly extends the expressiveness of above technique to permit the specification and automatic verification of various higher-order properties.

Note: We are including the below text to make this document self-contained; the bulk of this work originally appeared in [6].

1 Introduction

Refinement types offer an automatic means of verifying semantic properties of programs, by decorating types with predicates from logics efficiently decidable by modern SMT solvers. For example, the refinement type `{v: Int | v > 0}` denotes the basic type `Int` refined with a logical predicate over the "value variable" `v`. This type corresponds to the set of `Int` values `v` which additionally satisfy the logical predicate, *i.e.,* the set of positive integers. The (dependent) function type `x:{v:Int| v > 0} -> {v:Int| v < x}` describes functions that take a positive argument `x` and return an integer less than `x`.

Refinement type *checking* reduces to subtyping queries of the form $\Gamma \vdash \{\tau : \nu \mid p\} \preceq \{\tau : \nu \mid q\}$, where p and q are refinement predicates. These subtyping queries reduce to logical *validity* queries of the form $[\![\Gamma]\!] \wedge p \Rightarrow q$, which can be automatically discharged using SMT solvers [2]. Similarly, we have shown using the technique of Liquid Types [5] that refinement type *inference*, and hence, verification of higher-order programs, reduces to solving a system of *Horn-clauses* which are essentially the above implications with logical *variables* representing unknown (*i.e.,* to be inferred) refinements.

Unfortunately, the automatic verification offered by refinements has come at a price: to ensure decidable checking with SMT solvers, the refinements are quantifier-free predicates drawn from a decidable logic. This significantly limits expressiveness by precluding specifications that enable abstraction over the refinements (*i.e.,* invariants).

For example, consider the following higher-order for-loop:

```
for :: Int -> Int -> a -> (Int -> a -> a) -> a
for lo hi x body      = loop lo x
  where
    loop i x
          | i < hi    = loop (i+1) (body i x)
          | otherwise = x
```

Now, consider the following three *clients* of the `for` loop:

```
sum         :: Int -> Int
sum n       = for 0 n (\i tot -> i + tot)

range       :: Int -> Int -> Int
range n     = for 0 n (\i xs -> i : xs)

init        :: Vec a -> a -> Int -> Vec a
init a x n = for 0 n a (\i -> set i x)
```

Assume that `set i x v` returns the vector `v` updated at index `i` with the value `x`. Now, how would we verify that:

- `sum` returns a value that is larger than `n` ?
- `range` returns a decreasing list of integers between `n` and `0` ?
- `init` returns a vector whose first `n` elements are equal to `x`?

In a first-order setting, we would write (or infer) different *loop invariants* that described the machine's state at the $\texttt{i}^{th}$ iteration. For example, in `init`, an invariant stating the first `i` elements of the vector were already initalized to `x`.

However, in a higher-order setting we require a means of *abstracting* over possible invariants. That is, we require some way of *summarizing* the behavior of `for` that is *specific* enough to expose the loop index `i` but *generic* enough to apply to each of the three call-sites.

In this tutorial, we will describe a new technique called *abstract refinement types* which enable abstraction (quantification) over the refinements of data- and function-types. The main idea is that we can preserve SMT-based decidable analysis (checking and inference) by encoding abstract refinements as *uninterpreted* propositions in the refinement logic.

First, we illustrate how abstract refinements yield a variety of sophisticated means for reasoning about high-level program constructs, including: *parametric* refinements for type classes, *index-dependent* refinements for key-value maps, *recursive* refinements for data structures, and *inductive* refinements for higher-order traversal routines.

Second, we demonstrate that type checking remains decidable by showing a fully automatic procedure that uses SMT solvers, or to be precise, decision procedures based on congruence closure [4] to discharge logical subsumption queries over abstract refinements.

Third, we show that the crucial problem of *inferring* appropriate instantiations for the (abstract) refinement parameters boils down to inferring (non-abstract) refinement types, which we have previously automated via the abstract interpretation framework of Liquid Types [5].

Finally, we have implemented abstract refinements in LIQUIDHASKELL, a new Liquid Type-based verifier for Haskell. We will present a demo using LIQUIDHASKELL to concisely specify and verify a variety of correctness properties of several programs ranging from microbenchmarks to some widely used Haskell libraries.

2 Abstract Refinement Types

Abstract Refinement Types [6], are a means of enhancing expressiveness of a refinement system, while preserving (SMT-based) decidability. The key insight is that we avail quantification over the refinements of data- and function-types, simply by encoding refinement parameters as uninterpreted propositions within the refinement logic. We illustrate how this mechanism yields a variety of sophisticated means for reasoning about programs, including: inductive refinements for reasoning about higher-order traversal routines, compositional refinements for reasoning about function composition, index-dependent refinements for reasoning about key-value maps, and recursive refinements for reasoning about recursive data types.

2.1 The key idea

Consider the monomorphic `max` function on `Int` values. We give `max` a refinement type, stating that its result is greater or equal than both its arguments:

```
max     :: x:Int -> y:Int -> {v:Int | v >= x && v >= y}
max x y = if x > y then x else y
```

If we apply `max` to two positive integers, say `n` and `m`, we get that the result is greater or equal to both of them, as `max n m :: {v:Int | v >= n && v >= m}`. However, we can not reason about an arbitrary property: If we apply `max` to two even numbers, can not verify that the result is also even. Thus, even though we have the information that both arguments are even on the input, we lose it on the result.

To solve this problem, we introduce *abstract refinements* which let us quantify or parameterize a type over its constituent refinements. Using abstract refinements, we can type `max` as

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

where `Int<p>` is an abbreviation for the refinement type `{v:Int | p(v)}`. Intuitively, an abstract refinement `p` is encoded in the refinement logic as an *uninterpreted function symbol*, which satisfies the *congruence* axiom [4]

$$\forall \overline{X}, \overline{Y} : (\overline{X} = \overline{Y}) \Rightarrow P(\overline{X}) = P(\overline{Y})$$

It is trivial to verify, with an SMT solver, that `max` enjoys the above type: the input types ensure that both `p(x)` and `p(y)` hold and hence the returned value in either branch satisfies the refinement `{v:Int | p(v)}`, thereby ensuring the output type.

In a call site, we simply instantiate the *refinement* parameter of `max` with the concrete refinement, after which type checking proceeds as usual. As an example, suppose that we call `max` with two even numbers:

```
n :: {v:Int | even v}
m :: {v:Int | even v}
```

Then, the abstract refinement can be instantiated with a concrete predicate `even`, which will give `max` the type

```
max [even] ::
{v:Int | even v} -> {v:Int | even v} -> {v:Int | even v}
```

where the expression in brackets is the refinement instantiation. Since both `n` and `m` are even numbers, they satisfy the function's preconditions, thus we can apply them to `max`, to get an even result:

```
max [even] n m :: {v:Int |  even v}
```

This is the basic concept of abstract refinements, which as we shall see, have many interesting applications.

2.2 Function Composition

As a next example, we present how one can use abstract refinements to reason about function composition.

Consider a `plusminus` function that composes a plus and a minus operator:

```
plusminus :: n:Int
          -> m:Int
          -> x:Int
          -> {v:Int | v = (x - m) + n}
plusminus n m x = (x - m) + n
```

In a first order refinement system we can verify that the function's behaviour is captured by its type. However, consider an alternative definition that uses function composition `(.):: (b -> c)-> (a -> b)-> a -> c`.

```
plusminus n m x = plus . minus
  where plus  x = x + n
        minus x = x - m
```

It is unclear how to give `(.)` a (first-order) refinement type that expresses that the result can be refined with the composition of the refinements of both arguments results. Thus, this definition of `plusminus` can not have the previous descriptive type.

Typing function composition. To solve this problem, we can use abstract refinements and give `(.)` a type:

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

The trick is once again to quantify the type over refinements we care about. This time, we use two abstract refinements: the refinement `p` of the result of the first function `f` and the refinement `q` of the result of the second function `g`. For any argument `x`, we use an existential to bind the intermediate result to `z = g x`, so `z` satisfies `q` at `x`, and the result satisfies `p` at the intermediate result.

Using function composition. With this type for function composition, user functions get the concrete refinement of the final result to be the composition of the two refinements of the argument functions.

Back to the `plusminus` example, with the appropriate refinement instantiation we get the concrete refinement type for function composition:

```
(.) [{\x v -> v = x + n}, {\x v -> v = x - m}]
    :: f : (x:b -> {v:c | v = x+n})
    -> g : (x:a -> {v:b | v = x-m})
    -> x : a
    -> exists[z:{v:b | v = x-m}]. {v:c | v = z+n}
```

The result type asserts that there exists a value `z`, which is indeed the intermediate result, with the property `z = x - m`. With this, the final result is equal to `z + n`. If our logic supports equality, as SMT solvers do, we can verify that the final result is indeed equal to `(x - m)+ n`. In other words, we can verify the desired type of `plusminus`.

2.3 Inductive Refinements

As a first application we present how abstract refinements allow us to formalize induction within the type system.

Consider a `loop` function that takes as arguments a function `f`, an integer `n`, a base case `z` and applies the function `f` to the `z`, `n` times:

```
loop :: (Int -> a -> a) -> Int -> a -> a
loop f n z = go 0 z
  where go i acc | i < n      = go (i+1) (f i acc)
                 | otherwise = acc
```

Now consider a user function `incr` that uses `loop` and at each iteration increases the accumulator by one:

```
incr :: Int -> Int -> Int
incr n z = loop f n z
  where f i acc = acc + 1
```

The accumulator is initialized with `z` and at each `loop`'s iteration it is increased by `1`. So, at the `i`th iteration, the accumulator is equal to `z+i`. There will be `n` iterations, thus the final result will be `z+n`. This reasoning constitutes an inductive proof that characterizes `loop`'s behaviour. However, it is unclear how to give `loop` a (first-order) refinement type that describes its inductive behaviour. Hence, it has not been possible to verify that `incr` actually adds its two arguments.

Typing loop. Abstract refinements allow us to solve this problem, while remaining within the boundaries of SMT-based decidability. We give `loop` the following type:

```
loop :: forall <r :: Int -> a -> Bool> .
           f : (i:Int -> a<r i> -> a<r (i+1)>)
        -> n : {v:Int | n >= 0}
        -> z : a<r 0>
        -> a<r n>
```

The trick is to qualify over the invariant `r` that `loop` establishes between the loop iteration and the accumulator. Then the type signature encodes induction on natural numbers: (1) `n` should be a natural number, thus a non-negative integer, (2) the base case `z` should satisfy the invariant at 0, (3) in the inductive step, `f` uses the old accumulator to create the new one, thus if the old accumulator satisfies the invariant on the iteration `i`, the new one, as constructed by `f`, should satisfy the invariant at `i+1`. If these four conditions hold, we conclude that the result satisfies the invariant at `n`. This scheme is not novel [1]; what is new is the encoding, via uninterpreted predicate symbols in a SMT-decidable refinement type system.

Using loop. We can use this expressive type of `loop` to verify inductive properties of user functions:

```
incr :: n:{v:Int|v >= 0} -> z:Int -> {v:Int|v = n + z}
incr n z = loop [{\i acc -> acc + i}] f n z
  where f i acc = acc + 1
```

In the above example, the expression in brackets denotes the instantiation of the abstract refinement. For purpose of illustration we make abstract refinement instantiation explicit, but it could be automatically inferred via liquid typing [6].

2.4 Higher-Order Structure Traversals

Next, we generalize the previous approach and explain how abstract refinements allow us to formalize some kinds of structural induction within the type system.

Measures. First, let us formalize a notion of *length* for lists within the refinement logic. To do so, we define a special `len` measure by structural induction

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

We use the measures to automatically strengthen the types of the data constructors[3]:

```
data [a] where
  []  :: forall a.{v:[a] | len(v) = 0}
  (:) :: forall a.a -> xs:[a] -> {v:[a]|len(v)=1+len(xs)}
```

Note that the symbol `len` is encoded as an *uninterpreted* function in the refinement logic, and is, except for the congruence axiom, opaque to the SMT solver. The measures are guaranteed, by construction, to terminate, and so we can soundly use them as uninterpreted functions in the refinement logic. Notice also, that we can define *multiple* measures for a type; in this case we simply conjoin the refinements from each measure when refining each data constructor.

With these strengthened constructor types, we can verify, for example, that `append` produces a list whose length is the sum of the input lists' lengths:

```
append :: l:[a] -> m:[a] -> {v:[a]|len(v)=len(l)+len(m)}
append []     zs = zs
append (y:ys) zs = y : append ys zs
```

However, consider an alternate definition of `append` that uses `foldr`

```
append ys zs = foldr (:) zs ys
```

where `foldr :: (a -> b -> b)-> b -> [a] -> b`. It is unclear how to give `foldr` a (first-order) refinement type that captures the rather complex fact that the fold-function is "applied" all over the list argument, or, that it is a catamorphism. Hence, hitherto, it has not been possible to verify the second definition of `append`.

Typing Folds. Abstract refinements allow us to solve this problem with a very expressive type for `foldr` whilst remaining firmly within the boundaries of SMT-based decidability. We write a slightly modified fold:

```
foldr :: forall <p :: [a] -> b -> Bool>.
             (xs:[a] -> x:a -> b <p xs> -> <p (x:xs)>)
          -> b<p []>
          -> ys:[a]
          -> b<p ys>
foldr op b []     = b
foldr op b (x:xs) = op xs x (foldr op b xs)
```

The trick is simply to quantify over the relationship `p` that `foldr` establishes between the input list `xs` and the output `b` value. This is formalized by the type signature, which

encodes an induction principle for lists: the base value `b` must (1) satisfy the relation with the empty list, and the function `op` must take (2) a value that satisfies the relationship with the tail `xs` (we have added the `xs` as an extra "ghost" parameter to `op`), (3) a head value `x`, and return (4) a new folded value that satisfies the relationship with `x:xs`. If all the above are met, then the value returned by `foldr` satisfies the relation with the input list `ys`. This scheme is not novel in itself [1] — what is new is the encoding, via uninterpreted predicate symbols, in an SMT-decidable refinement type system.

Using Folds. Finally, we can use the expressive type for the above `foldr` to verify various inductive properties of client functions:

```
length :: zs:[a] -> {v: Int | v = len(zs)}
length = foldr (\_ _ n -> n + 1) 0

append :: l:[a] -> m:[a] -> {v:[a]| len(v)=len(l)+len(m)}
append ys zs = foldr (\_ -> (:)) zs ys
```

The verification proceeds by just (automatically) instantiating the refinement parameter `p` of `foldr` with the concrete refinements, via Liquid typing:

```
{\xs v -> v = len(xs)}                  -- for length
{\xs v -> len(v) = len(xs) + len(zs)}   -- for append
```

2.5 Index-Dependent Invariants

Next, we illustrate how abstract invariants allow us to specify and verify index-dependent invariants of key-value maps. To this end, we encode vectors as functions from `Int` to some generic range `a`. Formally, we specify vectors as

```
data Vec a <dom :: Int -> Bool, rng :: Int -> a -> Bool>
  = V (i:Int<dom> -> a <rng i>)
```

Here, we are parameterizing the definition of the type `Vec` with two abstract refinements, `dom` and `rng`, which respectively describe the *domain* and *range* of the vector. That is, `dom` describes the set of valid indices, and `rng` specifies an invariant relating each `Int` index with the value stored at that index.

Describing Vectors. With this encoding, we can describe various vectors. To start with we can have vectors of `Int` defined on positive integers with values equal to their index:

```
Vec <{\v -> v > 0}, {\_ v -> v = x}> Int
```

Or a vector that is defined only on index 1 with value 12:

```
Vec <{\v -> v = 1}, {\_ v -> v = 12}> Int
```

As a more interesting example, we can define a *Null Terminating String* with length `n`, as a vector of `Char` defined on a range `[0, n)` with its last element equal to the

terminating character:

```
Vec <{\v -> 0 <= v < n}
    ,{\i v -> i = n-1 => v = '\0'}> Char
```

Finally, we can encode a *Fibonacci memoization vector*, which can be used to efficiently compute a Fibonacci number, that is defined on positive integers and its value on index `i` is either zero or the `i`th Fibonacci number:

```
Vec <{\v -> 0 <= v}
    ,{\i v -> v != 0 => v = fib(i)}> Char
```

Using Vectors. A first step towards using vectors is to supply the appropriate types for vector operations, (e.g., set, get and empty). This usually means qualifying over the domain and the range of the vectors. Then, the programmer has to specify interesting vector properties, as we did for the Fibonacci memoization, or the null terminating string. Finally, the system can verify that user functions, that transform vectors, preserve these properties. This procedure is applied in [6], where, with the appropriate types for vector operations, we reason about functions that transform null terminating strings or efficiently compute a Fibonacci number.

2.6 Recursive Invariants

Finally, we describe how we use abstract refinements to reason about properties of recursive data structures. For the purpose of illustration, we define a refined version of a `List` datatype with values of type `a`:

```
data List a <p :: a -> a -> Bool>
  = N
  | C (hd :: a) (tl :: List <p> (a <p h>))
```

We are parametrizing the `List` over an abstract refinement `p` that relates two elements of type `a`. With this, the list is either the empty list `N`, or contains a head `hd` of type `a` and a tail `tl` which is a list of elements of type `a<p h>`, i.e., these elements satisfy the abstract refinement `p` at the head. Since this definition is recursively applied, the abstract refinement `p` holds between each pair of elements in the list.

Unfolding Lists. To demonstrate the previous property, we will unfold a `List` with three elements that satisfies an abstract refinement `p`. Consider such a list:

```
C h1 (C h2 (C h3 N)) :: List <p> a
```

If we unfold this list once, by the definition of the `C` data constructor, the first element is of type `a`, while the rest is a list with values that satisfy `p` at the first element, i.e., `(C h2 (C h3 N))::` `List <p> a<p h1>`. With a second unfold we get that the second element satisfies `p` at the first element, i.e., `h2::a<p h1>`, while the rest is a list with values that satisfy `p` at both the first and the second element, i.e., `C h3 N :: List <p> a<p h1 && p h2>`. With the last

unfold we get that the last element satisfies `p` at all the previous elements, i.e., `h3 :: a<p h1 && p h2>`, while the empty list satisfies `p` at every list element, i.e., `N :: List <p> a<p h1 && p h2 && p3>`, which holds as by its definition the empty list `N` satisfies any refinement.

Thus, `p` holds between every pair of the list, as for any two two elements `hi` and `hj`, with `i < j`, at the `i`th unfold `hj` satisfies `p` at `hi`.

If we instantiate the abstract refinement `p` with the concrete refinement `{\h v -> h <= v}`, that expresses that each values is greater than the head, we get that each element is greater than all its previous in the list. So we describe an increasing list:

```
type IncrList a = List <{\h v -> h <= v}> a
```

We can describe different list properties, by embedding appropriate concrete refinements. For instance, if we use a refinement that expresses that each value is less than the head, i.e., `{\h v -> h >= v}` or different from it, i.e., `{\h v -> h ~= v}`, we can describe decreasing or unique element lists.

Using Lists. We can use the refined type for lists to verify list properties. As an example, our system can verify that the following inserting sort algorithm actually returns an increasing list.

```
insertSort :: (Ord a) => [a] -> IncrList a
insertSort = foldr insert N

insert :: (Ord a) => a -> IncrList a -> IncrList a
insert y N        = C y N
insert y (C x xs) | y <= x         = C y (C x xs)
                  | otherwise      = C x (insert y xs)
```

Multiple Recursive Refinements. We can define recursive types with multiple parameters. For example, consider the following refined version of a type used to encode functional maps (`Data.Map`):

```
data Tree k v <l :: k->k->Bool, r :: k->k->Bool>
  = Bin { key   :: k
        , value :: v
        , left  :: Tree <l, r> (k <l key>) v
        , right :: Tree <l, r> (k <r key>) v }
  | Tip
```

The abstract refinements `l` and `r` relate each `key` of the tree with *all* the keys in the *left* and *right* subtrees of `key`, as those keys are respectively of type `k <l key>` and `k <r key>`. Thus, if we instantiate the refinements with the following predicates

```
type BST k v     = Tree<{\x y -> x> y},{\x y-> x< y}> k v
type MinHeap k v = Tree<{\x y -> x<=y},{\x y-> x<=y}> k v
type MaxHeap k v = Tree<{\x y -> x>=y},{\x y-> x>=y}> k v
```

then `BST k v`, `MinHeap k v` and `MaxHeap k v` denote exactly binary-search-ordered, min-heap-ordered, and max-heap-ordered trees (with keys and values of types `k` and `v`).

3 Plan

We will plan our tutorial as follows:

- *(10 mins, optional) Basic Refinement and Liquid Types* we will start with a rapid overview of refinement types and inference. Most likely audience members will be familiar with these ideas, perhaps through the invited talks (e.g. Jagannathan or Rybalchenko), and if so, we can skip this portion.
- *(30 mins) Abstract Refinements* We will work through the above examples to illustrate how program analysis is implemented as refinement type inference, which, in turn, reduces to Horn Clause solving. We will go over the above examples and demonstrate them in the tool. Since the tool and materials are online at:

  ```
  http://goto.ucsd.edu/~rjhala/liquid/haskell/demo/#?demo=absref101.hs
  http://goto.ucsd.edu/~rjhala/liquid/haskell/demo/#?demo=ListSort.hs
  http://goto.ucsd.edu/~rjhala/liquid/haskell/demo/#?demo=Map.hs
  http://goto.ucsd.edu/~rjhala/liquid/haskell/demo/#?demo=Foldr.hs
  ```

  it should be possible for the audience to follow along and modify as we go over the examples.
- *(10 mins, optional) Analyzing Libraries* We will demonstrate in how we use the above types to automatically verify ordering properties of complex, full-fledged libraries like `Data.Map` which use several of the above ideas.

References

1. Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development. Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. Springer Verlag, 2004.
2. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
3. M. Kawaguchi, P. Rondon, and R. Jhala. Type-based data structure verification. In *PLDI*, pages 304–315, 2009.
4. G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.
5. P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.
6. N. Vazou, P. Rondon, and R. Jhala. Abstract refinement types. In *ESOP 2013: European Symposium on Programming*. Springer, 2013.