Abstract Refinement Types

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

[1]UC San Diego [2]Google

Abstract. We present *abstract refinement types* which enable quantification over the refinements of data- and function-types. Our key insight is that we can avail of quantification while preserving SMT-based decidability, simply by encoding refinement parameters as *uninterpreted* propositions within the ground refinement logic. We illustrate how this simple mechanism yields a variety of sophisticated and automatic means for reasoning about programs, including: *parametric* refinements for reasoning with type classes, *index-dependent* refinements for reasoning about key-value maps, *recursive* refinements for reasoning about data structures, and *inductive* refinements for reasoning about higher-order traversal routines. We have implemented our approach in HSOLVE, a refinement type checker for Haskell, and present experiments using HSOLVE to verify correctness invariants of various programs.

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 $\Gamma \vdash \{\tau : \nu \mid p\} \prec \{\tau : \nu \mid q\}$, 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 [6].

Several groups have shown how refinement types can be used to verify properties ranging from partial correctness concerns like array bounds checking [26, 22] and data structure invariants [16] to the correctness of security protocols [2], web applications [14] and implementations of cryptographic protocols [10].

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

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

where `set i x v` returns the vector `v` updated at index `i` with the value `x`. We would like to verify that `initUpto` returns a vector whose *first* `n` elements are equal to `x`.

In a first-order setting, we could achieve the above with a loop invariant that asserted that at the $\mathtt{i}^{th}$ iteration, the first `i` elements of the vector were already initalized to `x`. However, in our higher-order setting we require a means of *abstracting* over possible invariants, each of which can *depend on* the iteration index `i`. Higher-order logics like Coq and Agda permit such quantification over invariants. Alas, validity in such logics is well outside the realm of decidability, and hence their use precludes automatic verification.

In this paper, we present *abstract refinement types* which enable abstraction (quantification) over the refinements of data- and function-types. Our key insight is that we can preserve SMT-based decidable type checking by encoding abstract refinements as *uninterpreted* propositions in the refinement logic. This yields several contributions:

- First, we illustrate how abstract refinements yield a variety of sophisticated means for reasoning about high-level program constructs (§2), 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 (§3), as SMT solvers can efficiently discharge logical subsumption queries over abstract refinements using decision procedures based on congruence closure [19]
- Third, we show that the crucial problem of *inferring* appropriate instantiations for the (abstract) refinement parameters boils down to inferring (non-abstract) refinement types (§3), which we have previously addressed via the abstract interpretation framework of Liquid Types [22].
- Finally, we have implemented abstract refinements in HSOLVE, a new Liquid Type-based verifier for Haskell. We present experiments using HSOLVE to concisely specify and automatically verify a variety of correctness properties of several programs ranging from microbenchmarks to some widely used libraries (§4).

2 Overview

We start with a high level overview of abstract refinements, by illustrating how they can be used to uniformly specify and automatically verify various kinds of invariants.

2.1 Parametric Invariants

Parametric Invariants via Type Polymorphism. Suppose we had a generic comparison operator of type `(<=):: a -> a -> Bool`, as is the case in OCAML. We could use it to write functions

```
max      :: a -> a -> a
max x y = if x <= y then y else x

maximum :: [a] -> a
maximum (x:xs) = foldr max x xs
```

In essence, the type given for `maximum` states that *for any* `a`, if a list of `a` values is passed into `maximum`, then the returned result is also an `a` value. Hence, for example, if a list of *prime* numbers is passed in, the result is prime, and if a list of *even* numbers is passed in, the result is even. Thus, we can use refinement types [22] to verify

```
type Even = {v:Int | v % 2 = 0 }

maxEvens :: [Int] -> Even
maxEvens xs = maximum (0 : xs')
  where xs' = [ x | x <- xs, x `mod` 2 = 0]
```

Here the `%` represents the modulus operator in the refinement logic [6] and we type the primitive `mod :: x:Int -> y:Int -> {v: Int | v = x % y}`. Verification proceeds as follows. Given that `xs :: [Int]`, the system has to verify that `maximum (0 : xs'):: Even`. To this end, the type parameter of `maximum` is instantiated with the *refined* type `Even`, yielding the instance:

```
maximum :: [Even] -> Even
```

Then, `maximum`'s argument should be proved to have type `[Even]`. So, the type parameter of `(:)` is instantiated with `Even`, yielding the instance:

```
(:) :: Even -> [Even] -> [Even]
```

Finally, the system infers that `0 :: Even` and `xs' :: [Even]`, *i.e.,* the arguments of `(:)` have the expected types, thereby verifying the program. The refined type instantiations can be *automatically* inferred, using the abstract interpretation framework of Liquid Types [22], with the appropriate set of qualifiers $\mathbb{Q}^\star$, thereby making the above verification fully automated.

Thus, classical type parameters offer an easy means of encoding second-order invariants, *i.e.,* of quantifying over or parametrizing the invariants of inputs and outputs of functions, without sacrificing automatic verification.

Parametric Invariants via Abstract Refinements. Instead, suppose that the comparison operator was monomorphic, and only worked for `Int` values. The resulting (monomorphic) signatures

```
max     :: Int -> Int -> Int
maximum :: [Int] -> Int
```

preclude the verification of maxEvens (*i.e.,* typechecking against the signature shown earlier). This is because the new type of maximum merely states that *some* Int is returned as output, and not necessarily one that enjoys the properties of the values in the input list. This is a shame, since the property clearly still holds.

We solve this problem with *abstract refinements*, which let us quantify or parameterize a type over its constituent refinements. For example, 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 [19]

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

Thus, 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. By the same reasoning, we can generalize the type of maximum to

```
maximum :: forall <p :: Int -> Bool>. [Int<p>] -> Int<p>
```

Consequently, we can recover the verification of maxEvens. Now, instead of instantiating a *type* parameter, we simply instantiate the *refinement* parameter of maximum with the concrete refinement {\v -> v % 2 = 0}, after which type checking proceeds as usual [22]. Later, we show how to retain automatic verification by inferring refinement parameter instantiations via liquid typing (§ 3.4).

Parametric Invariants and Type Classes. The example above regularly arises in practice, due to type classes. In Haskell, the functions above are typed

```
(<=)    :: (Ord a) => a -> a -> Bool
max     :: (Ord a) => a -> a -> a
maximum :: (Ord a) => [a] -> a
```

We might be tempted to simply ignore the type class constraint, and just treat maximum as [a] -> a but, of course, this would be quite unsound, as typeclass predicates trivially preclude universal quantification over refinement types. Consider the function sum :: (**Num** a)=> [a] -> a which adds the elements of a list. The **Num** class constraint implies that numeric operations occur in the function, so if we pass sum a list of odd numbers, we are *not* guaranteed to get back an odd number.

Thus, how do we soundly verify the desired type of maxEvens without instantiating class predicated type parameters with arbitrary refinement types? First, via the same analysis as the monomorphic Int case, we establish that

```
max:: forall <p::a->Bool>. (Ord a)=> a<p> -> a<p> -> a<p>
maximum:: forall <p::a ->Bool>. (Ord a) => [a<p>] -> a<p>
```

Next, at the call-site for maximum in maxEvens we instantiate the type variable a with Int, and the abstract refinement p with {\v -> v % 2 = 0} after which, the verification proceeds as described earlier (for the `Int` case). Thus, abstract refinements allow us to quantify over invariants without relying on parametric polymorphism, even in the presence of type classes.

2.2 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 develop a small library of *extensible vectors* encoded, for purposes of illustration, 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 `r` specifies an invariant relating each `Int` index with the value stored at that index.

Creating Vectors. We can use the following basic functions to create vectors:

```
empty :: forall <p::Int->a->Bool>.Vec<{\_ -> False}, p> a
empty = V (\_ -> error "Empty Vec")

create :: x:a -> Vec <{\_ -> True}, {\_ v -> v = x}> a
create x = V (\_ -> x)
```

The signature for `empty` states that its domain is empty (*i.e.,* is the set of indices satisfying the predicate **`False`**), and that the range satisfies `any` invariant. The signature for `create`, instead, defines a *constant* vector that maps every index to the constant `x`.

Accessing Vectors. We can write the following `get` function for reading the contents of a vector at a given index:

```
get :: forall <d :: Int -> Bool, r :: Int -> a -> Bool>
          i:Int<d> -> Vec<d, r> a -> a<r i>
get i (V f) = f i
```

The signature states that for any domain `d` and range `r`, if the index `i` is a valid index, *i.e.,* is of type, `Int<d>` then the returned value is an `a` that additionally satisfies the range refinement at the index `i`. The type for `set`, which *updates* the vector at a given index, is even more interesting, as it allows us to *extend* the domain of the vector:

```
set :: forall <d :: Int -> Bool, r :: Int -> a -> Bool>
          i:Int<d>
       -> a<r i>
       -> Vec<d && {\k -> k != i}, r> a
       -> Vec<d, r> a
set i v (V f) = V (\k -> if k == i then x else f k)
```

The signature for `set` requires that (a) the input vector is defined everywhere at `d` *except* the index `i`, and (b) the value supplied must be of type `a<r i>`, *i.e.,* satisfy the range relation at the index `i` at which the vector is being updated. The signature ensures that the output vector is defined at `d` and each value satisfies the index-dependent range refinement `r`. Note that it is legal to call `get` with a vector that is *also* defined at the index `i` since, by contravariance, such a vector is a subtype of that required by (a).

Initializing Vectors. Next, we can write the following function, `init`, that "loops" over a vector, to `set` each index to a value given by some function.

```
initialize :: forall <r :: Int -> a -> Bool>.
               (z: Int -> a<r z>)
            -> i: {v: Int | v >= 0}
            -> n: Int
            -> Vec <{\v -> 0 <= v && v < i}, r> a
            -> Vec <{\v -> 0 <= v && v < n}, r> a

initialize f i n a
  | i >= n    = a
  | otherwise = initialize f (i+1) n (set i (f i) a)
```

The signature requires that (a) the higher-order function `f` produces values that satisfy the range refinement `r`, and (b) the vector is initialized from `0` to `i`. The function ensures that the output vector is initialized from `0` through `n`. We can thus verify that

```
idVec   :: Vec <{\v -> 0<=v && v<n}, {\i v -> v=i}> Int
idVec n = initialize (\i -> i) 0 n empty
```

i.e., `idVec` returns an vector of size `n` where each key is mapped to itself. Thus, abstract refinement types allow us to verify low-level idioms such as the incremental initialization of vectors, which have previously required special analyses [12, 15, 5].

Null-Terminated Strings. We can also use abstract refinements to verify code which manipulates C-style null-terminated strings, represented as `Char` vectors for ease of exposition. Formally, a null-terminated string of size `n` has the type

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

The above type describes a length-`n` vector of characters whose last element must be a null character, signalling the end of the string. We can use this type in the specification of a function, `upperCase`, which iterates through the characters of a string, uppercasing each one until it encounters the null terminator:

```
upperCase :: n:{v: Int| v>0} -> NullTerm n -> NullTerm n
upperCase n s = ucs 0 s where
  ucs i s = case get i s of
              '\0' -> s
              c    -> ucs (i + 1) (set i (toUpper c) s)
```

Note that the length parameter `n` is provided solely as a "witness" for the length of the string `s`, which allows us to use the length of `s` in the type of `upperCase`; `n` is not used in the computation. In order to establish that each call to `get` accesses string `s` within its bounds, our type system must establish that, at each call to the inner function `ucs`, `i` satisfies the type `{v: Int | 0 <= v && v < n}`. This invariant is established as follows. First, the invariant trivially holds on the first call to `ucs`, as `n` is positive and `i` is 0. Second, we assume that `i` satisfies the type `{v: Int | 0 <= v && v < n}`, and, further, we know from the types of `s` and `get` that `c` has the type `{v: Char | i = n - 1 => c = '\0'}`. Thus, if `c` is non-null, then `i` cannot be equal to `n - 1`. This allows us to strengthen our type for `i` in the else branch to `{v: Int | 0 <= v && v < n - 1}` and thus to conclude that the value `i + 1` recursively passed as the `i` parameter to `ucs` satisfies the type `{v: Int | 0 <= v && v < n}`, establishing the inductive invariant and thus the safety of the `upperCase` function.

Memoization. Next, let us illustrate how the same expressive signatures allow us to verify memoizing functions. We can specify to the SMT solver the definition of the Fibonacci function via an uninterpreted function `fib` and an axiom:

```
measure fib :: Int -> Int
axiom: forall i. fib(i) = i<=1 ? 1 : fib(i-1) + fib(i-2)
```

Next, we define a type alias `FibV` for the vector whose values are either `0` (*i.e.,* undefined), or equal to the Fibonacci number of the corresponding index.

```
type FibV = Vec<{\_->True},{\i v-> v!=0 => v=fib(i)}> Int
```

Finally, we can use the above alias to verify `fastFib`, an implementation of the Fibonacci function, which uses an vector memoize intermediate results

```
fastFib :: n:Int -> {v:Int | v = fib(n)}
fastFib n = snd $ fibMemo (create 0) n

fibMemo :: FibV -> i:Int -> (FibV, {v: Int | v = fib(i)})
fibMemo t i
  | i <= 1    = (t, 1)
  | otherwise = case get i t of
                  0 -> let (t1, n1) = fibMemo t  (i-1)
                           (t2, n2) = fibMemo t1 (i-2)
                           n        = n1 + n2
                       in  (set i n t2,  n)
                  n -> (t, n)
```

Thus, abstract refinements allow us to define key-value maps with index-dependent refinements for the domain and range. Quantification over the domain and range refinements allows us to define generic access operations (*e.g.,* `get`, `set`, `create`, `empty`) whose types enable us establish a variety of precise invariants.

2.3 Recursive Invariants

Next, we turn our attention to recursively defined datatypes, and show how abstract refinements allow us to specify and verify high-level invariants that relate the elements of a recursive structure. Consider the following refined definition for lists:

```
data [a] <p :: a -> a -> Bool> where
  []  :: [a]<p>
  (:) :: h:a -> [a<p h>]<p> -> [a]<p>
```

The definition states that a value of type `[a]<p>` is either empty (`[]`) or constructed from a pair of a *head* `h::a` and a *tail* of a list of `a` values *each* of which satisfies the refinement `(p h)`. Furthermore, the abstract refinement `p` holds recursively within the tail, ensuring that the relationship `p` holds between *all* pairs of list elements.

Thus, by plugging in appropriate concrete refinements, we can define the following aliases, which correspond to the informal notions implied by their names:

```
type IncrList a = [a]<{\h v -> h <= v}>
type DecrList a = [a]<{\h v -> h >= v}>
type UniqList a = [a]<{\h v -> h != v}>
```

That is, `IncrList a` (resp. `DecrList a`) describes a list sorted in increasing (resp. decreasing) order, and `UniqList a` describes a list of *distinct* elements, *i.e.,* not containing any duplicates. We can use the above definitions to verify

```
[1, 2, 3, 4] :: IncrList Int
[4, 3, 2, 1] :: DecrList Int
[4, 1, 3, 2] :: UniqList Int
```

More interestingly, we can verify that the usual algorithms produce sorted lists:

```
insertSort :: (Ord a) => [a] -> IncrList a
insertSort []     = []
insertSort (x:xs) = insert x (insertSort xs)

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

Thus, abstract refinements allow us to *decouple* the definition of the list from the actual invariants that hold. This, in turn, allows us to conveniently reuse the same underlying (non-refined) type to implement various algorithms unlike, say, singleton-type based implementations which require up to three different types of lists (with three different "nil" and "cons" constructors [23]). This, makes abstract refinements convenient for verifying complex sorting implementations like that of `Data.List.sort` which, for efficiency, use lists with different properties (*e.g.,* increasing and decreasing).

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`). We demonstrate in (§ 4) how we use the above types to automatically verify ordering properties of complex, full-fledged libraries.

2.4 Inductive Invariants

Finally, we 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[16]:

```
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 [3] — 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
```

Expressions	$e ::= x \mid c \mid \lambda x:\tau.e \mid e\ e \mid \Lambda\alpha.e \mid e\,[\tau] \mid \Lambda\pi:\tau.e \mid e\,[e]$
Abstract Refinements	$p ::= \mathit{true} \mid p \wedge \pi\ \overline{e}$
Basic Types	$b ::= \texttt{int} \mid \texttt{bool} \mid \alpha$
Abstract Refinement Types	$\tau ::= \{v : b\langle p\rangle \mid e\} \mid \{v : (x:\tau) \rightarrow \tau \mid e\}$
Abstract Refinement Schemas	$\sigma ::= \tau \mid \forall\alpha.\sigma \mid \forall\pi:\tau.\sigma$

Fig. 1. Syntax of Expressions, Refinements, Types and Schemas

3 Syntax and Semantics

Next, we present a core calculus λ_P that formalizes the notion of abstract refinements. We start with the syntax (§ 3.1), present the typing rules (§ 3.2), show soundness via a reduction to contract calculi [17, 1] (§ 3.3), and inference via Liquid types (§ 3.4).

3.1 Syntax

Figure 1 summarizes the syntax of our core calculus λ_P which is a polymorphic λ-calculus extended with abstract refinements.

Expressions. λ_P expressions include the standard variables x, primitive constants c, λ-abstraction $\lambda x:\tau.e$, application $e\ e$, type abstraction $\Lambda\alpha.e$, and type application $e\,[\tau]$. The parameter τ in the type application is a *refinement type*, as described shortly. The two new additions to λ_P are the refinement abstraction $\Lambda\pi:\tau.e$, which introduces a refinement variable π (together with its type τ), which can appear in refinements inside e, and the corresponding refinement application $e\,[e]$.

Refinements. A *concrete refinement* e is a boolean valued expression e (which we will embed into an SMT decidable refinement logic including the theory of linear arithmetic and uninterpreted functions.) An *abstract refinement* p is a conjunction of refinement variable applications of the form $\pi\ \overline{e}$.

Types and Schemas. The basic types of λ_P include the base types `int` and `bool` and type variables α. An *abstract refinement type* τ is either a basic type refined with an abstract and concrete refinements, $\{v : b\langle p\rangle \mid e\}$, or a dependent function type where the parameter x can appear in the refinements of the output type. We include refinements for functions, as refined type variables can be replaced by function types. However, typechecking ensures these refinements are trivially true. Finally, types can be quantified over refinement variables and type variables to yield abstract refinement schemas.

Notation. We write b, $\{v : b \mid e\}$ and $b\langle p\rangle$ to abbreviate $\{v : b\langle \mathit{true}\rangle \mid \mathit{true}\}$, $\{v : b\langle \mathit{true}\rangle \mid e\}$, and $\{v : b\langle p\rangle \mid \mathit{true}\}$ respectively. We say a type or schema is *non-refined* if all the refinements in it are *true*. We write $\overline{z}$ to abbreviate a sequence $z_1 \ldots z_n$.

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

$$\frac{}{\Gamma \vdash \mathit{true}(v)}\ \text{WF-True} \qquad \frac{\Gamma \vdash p(v) \quad \Gamma \vdash \pi\ \overline{e}\ v : \texttt{bool}}{\Gamma \vdash (p \wedge \pi\ \overline{e})(v)}\ \text{WF-RApp}$$

$$\frac{\Gamma, v : b \vdash e : \texttt{bool} \quad \Gamma, v : b \vdash p(v) : \texttt{bool}}{\Gamma \vdash \{v : b\langle p \rangle \mid e\}}\ \text{WF-Base}$$

$$\frac{\Gamma \vdash e : \texttt{bool} \quad \Gamma \vdash \tau_x \quad \Gamma, x : \tau_x \vdash \tau}{\Gamma \vdash \{v : (x : \tau_x) \rightarrow \tau \mid e\}}\ \text{WF-Fun}$$

$$\frac{\Gamma, \pi : \tau \vdash \sigma}{\Gamma \vdash \forall \pi : \tau . \sigma}\ \text{WF-Abs-}\pi \qquad \frac{\Gamma, \alpha \vdash \sigma}{\Gamma \vdash \forall \alpha . \sigma}\ \text{WF-Abs-}\alpha$$

Subtyping $\boxed{\Gamma \vdash \sigma_1 \prec \sigma_2}$

$$\frac{\text{Valid}([\![\Gamma]\!] \wedge [\![p_1\ v]\!] \wedge [\![e_1]\!] \Rightarrow [\![p_2\ v]\!] \wedge [\![e_2]\!])}{\Gamma \vdash \{v : b\langle p_1 \rangle \mid e_1\} \prec \{v : b\langle p_2 \rangle \mid e_2\}}\ \prec\text{-Base}$$

$$\frac{\Gamma \vdash \tau_2 \prec \tau_1 \quad \Gamma, x_2 : \tau_2 \vdash \tau_1'[x_2/x_1] \prec \tau_2'}{\Gamma \vdash \{v : (x_1 : \tau_1) \rightarrow \tau_1' \mid e_1\} \prec \{v : (x_2 : \tau_2) \rightarrow \tau_2' \mid \mathit{true}\}}\ \prec\text{-Fun}$$

$$\frac{\Gamma, \pi : \tau \vdash \sigma_1 \prec \sigma_2}{\Gamma \vdash \forall \pi : \tau . \sigma_1 \prec \forall \pi : \tau . \sigma_2}\ \prec\text{-RVar} \qquad \frac{\Gamma \vdash \sigma_1 \prec \sigma_2}{\Gamma \vdash \forall \alpha . \sigma_1 \prec \forall \alpha . \sigma_2}\ \prec\text{-Poly}$$

Type Checking $\boxed{\Gamma \vdash e : \sigma}$

$$\frac{\Gamma \vdash e : \sigma_2 \quad \Gamma \vdash \sigma_2 \prec \sigma_1 \quad \Gamma \vdash \sigma_1}{\Gamma \vdash e : \sigma_1}\ \text{T-Sub} \qquad \frac{}{\Gamma \vdash c : tc\,(c)}\ \text{T-Const}$$

$$\frac{x : \{v : b\langle p \rangle \mid e\} \in \Gamma}{\Gamma \vdash x : \{v : b\langle p \rangle \mid e \wedge v = x\}}\ \text{T-Var-Base} \qquad \frac{x : \tau \in \Gamma}{\Gamma \vdash x : \tau}\ \text{T-Var}$$

$$\frac{\Gamma, x : \tau_x \vdash e : \tau \quad \Gamma \vdash \tau_x}{\Gamma \vdash \lambda x : \tau_x . e : (x : \tau_x) \rightarrow \tau}\ \text{T-Fun} \qquad \frac{\Gamma \vdash e_1 : (x : \tau_x) \rightarrow \tau \quad \Gamma \vdash e_2 : \tau_x}{\Gamma \vdash e_1\ e_2 : \tau[e_2/x]}\ \text{T-App}$$

$$\frac{\Gamma, \alpha \vdash e : \sigma}{\Gamma \vdash \Lambda \alpha . e : \forall \alpha . \sigma}\ \text{T-Gen} \qquad \frac{\Gamma \vdash e : \forall \alpha . \sigma \quad \Gamma \vdash \tau}{\Gamma \vdash e\,[\tau] : \sigma[\tau/\alpha]}\ \text{T-Inst}$$

$$\frac{\Gamma, \pi : \tau \vdash e : \sigma \quad \tau \text{ is non-refined}}{\Gamma \vdash \Lambda \pi : \tau . e : \forall \pi : \tau . \sigma}\ \text{T-PGen} \qquad \frac{\Gamma \vdash e : \forall \pi : \tau . \sigma \quad \Gamma \vdash \lambda \overline{x : \tau_x} . e' : \tau}{\Gamma \vdash e\,[\lambda \overline{x : \tau_x} . e'] : \sigma[\pi \rhd \lambda \overline{x : \tau_x} . e']}\ \text{T-PInst}$$

Fig. 2. Static Semantics: Well-formedness, Subtyping and Type Checking

3.2 Static Semantics

Next, we describe the static semantics of λ_P by describing the typing judgments and derivation rules. Most of the rules are standard [21, 22, 17, 2]; we discuss only those pertaining to abstract refinements.

Judgments. A type environment Γ is a sequence of type bindings $x : \sigma$. We use environments to define three kinds of typing judgments:

- ***Wellformedness judgments*** ($\Gamma \vdash \sigma$) state that a type schema σ is well-formed under environment Γ, that is, the refinements in σ are boolean expressions in the environment Γ.
- ***Subtyping judgments*** ($\Gamma \vdash \sigma_1 \prec \sigma_2$) state that the type schema σ_1 is a subtype of the type schema σ_2 under environment Γ, that is, when the free variables of σ_1 and σ_2 are bound to values described by Γ, the set of values described by σ_1 is contained in the set of values described by σ_2.
- ***Typing judgments*** ($\Gamma \vdash e : \sigma$) state that the expression e has the type schema σ under environment Γ, that is, when the free variables in e are bound to values described by Γ, the expression e will evaluate to a value described by σ.

Wellformedness Rules. The wellformedness rules check that the concrete and abstract refinements are indeed `bool`-valued expressions in the appropriate environment. The key rule is WF-BASE, which checks, as usual, that the (concrete) refinement e is boolean, and additionally, that the abstract refinement p applied to the value v is also boolean. This latter fact is established by WF-RAPP which checks that each refinement variable application $\pi\ \overline{e}\ v$ is also of type `bool` in the given environment.

Subtyping Rules. The subtyping rules stipulate when the set of values described by schema σ_1 is subsumed by the values described by σ_2. The rules are standard except for $\prec$-VAR, which encodes the base types' abstract refinements p_1 and p_2 with conjunctions of *uninterpreted predicates* $[\![p_1\ v]\!]$ and $[\![p_2\ v]\!]$ in the refinement logic as follows:

$$\begin{aligned} [\![\mathit{true}\ v]\!] &\doteq \mathit{true} \\ [\![(p \wedge \pi\ \overline{e})\ v]\!] &\doteq [\![p\ v]\!] \wedge \pi([\![e_1]\!], \ldots, [\![e_n]\!], v) \end{aligned}$$

where $\pi(\overline{e})$ is a term in the refinement logic corresponding to the application of the uninterpreted predicate symbol π to the arguments $\overline{e}$.

Type Checking Rules. The type checking rules are standard except for T-PGEN and T-PINST, which pertain to abstraction and instantiation of abstract refinements. The rule T-PGEN is the same as T-FUN: we simply check the body e in the environment extended with a binding for the refinement variable π. The rule T-PINST checks that the concrete refinement is of the appropriate (unrefined) type τ, and then replaces all (abstract) applications of π inside σ with the appropriate (concrete) refinement e' with the parameters $\overline{x}$ replaced with arguments at that application. Formally, this is represented as $\sigma[\pi \rhd \lambda\overline{x:\tau}.e']$ which is σ with each base type transformed as

$$\begin{aligned} \{v : b\langle p\rangle \mid e\}[\pi \rhd z] &\doteq \{v : b\langle p''\rangle \mid e \wedge e''\} \\ \text{where}\quad (p'', e'') &\doteq \mathsf{Apply}(p, \pi, z, \mathit{true}, \mathit{true}) \end{aligned}$$

Apply replaces each application of π in p with the corresponding conjunct in e'', as

$$\begin{aligned} \mathsf{Apply}(\mathit{true}, \cdot, \cdot, p', e') &\doteq (p', e') \\ \mathsf{Apply}(p \wedge \pi'\ \overline{e}, \pi, z, p', e') &\doteq \mathsf{Apply}(p, \pi, z, p' \wedge \pi'\ \overline{e}, e') \\ \mathsf{Apply}(p \wedge \pi\ \overline{e}, \pi, \lambda\overline{x:\tau}.e'', p', e') &\doteq \mathsf{Apply}(p, \pi, \lambda\overline{x:\tau}.e'', p', e' \wedge e''[\overline{e}, v/\overline{x}]) \end{aligned}$$

In other words, the instantiation can be viewed as two symbolic reduction steps: first replacing the refinement variable with the concrete refinement, and then "beta-reducing" concrete refinement with the refinement variable's arguments. For example,

$$\{v : \texttt{int}\langle\pi\ y\rangle \mid v > 10\}[\pi \rhd \lambda x_1 : \tau_1.\lambda x_2 : \tau_2.x_1 < x_2] \doteq \{v : \texttt{int} \mid v > 10 \wedge y < v\}$$

3.3 Soundness

As hinted by the discussion about refinement variable instantiation, we can intuitively think of abstract refinement variables as *ghost* program variables whose values are boolean-valued functions. Hence, abstract refinements are a special case of higher-order contracts, that can be statically verified using uninterpreted functions. (Since we focus on static checking, we don't care about the issue of blame.) We formalize this notion by translating λ_P programs into the contract calculus $\mathrm{F_H}$ of [1] and use this translation to define the dynamic semantics and establish soundness.

Translation. We translate λ_P schemes σ to $\mathrm{F_H}$ schemes $\langle\!|\sigma|\!\rangle$ as by translating abstract refinements into contracts, and refinement abstraction into function types:

$$\begin{aligned} \langle\!|\mathit{true}\ v|\!\rangle &\doteq \mathit{true} & \langle\!|\forall\pi:\tau.\sigma|\!\rangle &\doteq (\pi : \langle\!|\tau|\!\rangle) \rightarrow \langle\!|\sigma|\!\rangle \\ \langle\!|(p \wedge \pi\ \overline{e})\ v|\!\rangle &\doteq \langle\!|p\ v|\!\rangle \wedge \pi\ \overline{e}\ v & \langle\!|\forall\alpha.\sigma|\!\rangle &\doteq \forall\alpha.\langle\!|\sigma|\!\rangle \\ \langle\!|\ \{v : b\langle p\rangle \mid e\}\ |\!\rangle &\doteq \{v : b \mid e \wedge \langle\!|p\ v|\!\rangle\} & \langle\!|(x:\tau_1) \rightarrow \tau_2|\!\rangle &\doteq (x : \langle\!|\tau_1|\!\rangle) \rightarrow \langle\!|\tau_2|\!\rangle \end{aligned}$$

Similarly, we translate λ_P terms e to $\mathrm{F_H}$ terms $\langle\!|e|\!\rangle$ by converting refinement abstraction and application to λ-abstraction and application

$$\begin{aligned} \langle\!|x|\!\rangle &\doteq x & \langle\!|c|\!\rangle &\doteq c \\ \langle\!|\lambda x:\tau.e|\!\rangle &\doteq \lambda x : \langle\!|\tau|\!\rangle.\langle\!|e|\!\rangle & \langle\!|e_1\ e_2|\!\rangle &\doteq \langle\!|e_1|\!\rangle\ \langle\!|e_2|\!\rangle \\ \langle\!|\Lambda\alpha.e|\!\rangle &\doteq \Lambda\alpha.\langle\!|e|\!\rangle & \langle\!|e\ [\tau]\ |\!\rangle &\doteq \langle\!|e|\!\rangle\ \langle\!|\tau|\!\rangle \\ \langle\!|\Lambda\pi:\tau.e|\!\rangle &\doteq \lambda\pi : \langle\!|\tau|\!\rangle.\langle\!|e|\!\rangle & \langle\!|e_1\ [e_2]\ |\!\rangle &\doteq \langle\!|e_1|\!\rangle\ \langle\!|e_2|\!\rangle \end{aligned}$$

Translation Properties. We can show by induction on the derivations that the type derivation rules of λ_P *conservatively approximate* those of $\mathrm{F_H}$. Formally,

- If $\Gamma \vdash \tau$ then $\langle\!|\Gamma|\!\rangle \vdash_H \langle\!|\tau|\!\rangle$,
- If $\Gamma \vdash \tau_1 \prec \tau_2$ then $\langle\!|\Gamma|\!\rangle \vdash_H \langle\!|\tau_1|\!\rangle <: \langle\!|\tau_2|\!\rangle$,
- If $\Gamma \vdash e : \tau$ then $\langle\!|\Gamma|\!\rangle \vdash_H \langle\!|e|\!\rangle : \langle\!|\tau|\!\rangle$.

Soundness. Thus rather than re-prove preservation and progress for λ_P, we simply use the fact that the type derivations are conservative to derive the following preservation and progress corollaries from [1]:

- **Preservation:** If $\emptyset \vdash e : \tau$ and $\langle\!|e|\!\rangle \longrightarrow e'$ then $\emptyset \vdash_H e' : \langle\!|\tau|\!\rangle$
- **Progress:** If $\emptyset \vdash e : \tau$, then either $\langle\!|e|\!\rangle \longrightarrow e'$ or $\langle\!|e|\!\rangle$ is a value.

Note that, in a contract calculus like $\mathrm{F_H}$, subsumption is encoded as a *upcast*. However, if subtyping relation can be statically guaranteed (as is done by our conservative SMT based subtyping) then the upcast is equivalent to the identity function and can be eliminated. Hence, $\mathrm{F_H}$ terms $\langle\!|e|\!\rangle$ translated from well-typed λ_P terms e have no casts.

3.4 Refinement Inference

Our design of abstract refinements makes it particularly easy to perform type inference via Liquid typing, which is crucial for making the system usable by eliminating the tedium of instantiating refinement parameters all over the code. (With value-dependent refinements, one cannot simply use, say, unification to determine the appropriate instantiations, as is done for classical type systems.) We briefly recall how Liquid types work, and sketch how they are extended to infer refinement instantiations.

Liquid Types. The Liquid Types method infers refinements in three steps. First, we create refinement *templates* for the unknown, to-be-inferred refinement types. The *shape* of the template is determined by the underlying (non-refined) type it corresponds to, which can be determined from the language's underlying (non-refined) type system. The template is just the shape refined with fresh refinement variables κ denoting the unknown refinements at each type position. For example, from a type $(x : \texttt{int}) \rightarrow \texttt{int}$ we create the template $(x : \{v : \texttt{int} \mid \kappa_x\}) \rightarrow \{v : \texttt{int} \mid \kappa\}$. Second, we perform type checking using the templates (in place of the unknown types.) Each wellformedness check becomes a wellformedness constraint over the templates, and hence over the individual κ, constraining which variables can appear in κ. Each subsumption check becomes a subtyping constraint between the templates, which can be further simplified, via syntactic subtyping rules, to a logical implication query between the variables κ. Third, we solve the resulting system of logical implication constraints (which can be cyclic) via abstract interpretation — in particular, monomial predicate abstraction over a set of logical qualifiers [9, 22]. The solution is a map from κ to conjunctions of qualifiers, which, when plugged back into the templates, yields the inferred refinement types.

Inferring Refinement Instantiations. The key to making abstract refinements practical is a means of synthesizing the appropriate arguments e' for each refinement application $e\,[e']$. Note that for such applications, we can, from e, determine the non-refined type of e', which is of the form $\tau_1 \rightarrow \ldots \rightarrow \tau_n \rightarrow \texttt{bool}$. Thus, e' has the template $\lambda x_1 : \tau_1 \ldots . \lambda x_n : \tau_n . \kappa$ where κ is a fresh, unknown refinement variable that must be solved to a boolean valued expression over $x_1, \ldots, x_n$. Thus, we generate a *well-formedness* constraint $x_1 : \tau_1, \ldots, x_n : \tau_n \vdash \kappa$ and carry out typechecking with template, which, as before, yields implication constraints over κ, which can, as before, be solved via predicate abstraction. Finally, in each refinement template, we replace each κ with its solution e_κ to get the inferred refinement instantiations.

4 Evaluation

In this section, we empirically evaluate the expressiveness and usability of abstract refinement types by exploring the process of typechecking a set of challenging benchmark programs using a prototype type checker for Haskell. (We defer the task of extending the metatheory to a call-by-name calculus to future work.)

HSOLVE. We have implemented abstract refinement in HSOLVE, a refinement type checker for Haskell. HSOLVE verifies Haskell source one module (.hs file) at a time. It takes as input:

Program	**LOC**	**Specs**	**Annot**	**Time (s)**
`Micro`	32	19	4	2
`Vector`	33	53	0	5
`ListSort`	29	4	1	3
`Data.List.sort`	71	3	1	8
`Data.Set.Splay`	136	13	11	13
`Data.Map.Base`	1395	128	24	136
Total	1696	220	41	167

Table 1. (LOC) is the number of non-comment Haskell source code lines as reported by *sloccount*, **(Specs)** is the number of lines of type specifications, **(Annot)** is the number of lines of other annotations, including refined datatype definitions, type aliases and measures, required for verification, **(Time)** is the time in seconds taken for verification.

- A *target* Haskell source file, with the desired refinement types specified as a special form of comment annotation,
- An (optional) set of type specifications for imported definitions; these can either be put directly in the source for the corresponding modules, if available, or in special `.spec` files otherwise. For imported functions for which no signature is given, HSOLVE conservatively uses the non-refined Haskell type.
- An (optional) set of logical qualifiers, which are predicate templates from which refinements are automatically synthesized [22]. Formally, a logical qualifier is a predicate whose variables range over the program variables, the special value variable ν, and *wildcards* $\star$, which HSOLVE instantiates with the names of program variables. Aside from the qualifiers given by the user, HSOLVE also uses qualifiers mined from the refinement type annotations present in the program.

After analyzing the program, HSOLVE returns as output:

- Either SAFE, indicating that all the specifications indeed verify, or UNSAFE, indicating there are refinement type errors, together with the positions in the source code where type checking fails (*e.g.,* functions that do not satisfy their signatures, or callsites where the inputs don't conform to the specifications).
- An HTML file containing the program source code annotated with inferred refinement types for all sub-expressions in the program. The inferred refinement type for each program expression is the strongest possible type over the given set of logical qualifiers. When a type error is reported, the programmer can use the inferred types to determine why their program does not typecheck: they can examine what properties HSOLVE *can* deduce about various program expressions and add more qualifiers or alter the program as necessary so that it typechecks.

Implementation. HSOLVE verifies the contents of a single file (module) at a time as follows. First, the Haskell source is fed into GHC, which desugars the program to GHC's "core" intermediate representation [25]. Second, the desugared program, the type signatures for the module functions (which are to be verified) and the type signatures for externally imported functions (which are assumed to hold) are sent to the constraint generator, which traverses the core bindings in a syntax-directed manner to generate subtyping constraints. The resulting constraints are simplified via our subtyping rules (§ 3) into simple logical implication constraints. Finally, the implication constraints, together with the logical qualifiers provided by the user and harvested from the type signatures, are sent into an SMT- and abstract interpretation-based fixpoint computation procedure that determines if the constraints are satisfiable [13, 9]. If so, the program is reported to be *safe*. Otherwise, each unsatisfiable constraint is mapped back to the corresponding program source location that generated it and a potential error is reported at that line in the program.

Benchmarks. We have evaluated HSOLVE over the following list of benchmarks which, in total, represent the different kinds of reasoning described in § 2. While we can prove, and previously have proved [16], many so-called "functional correctness" properties of these data structures using refinement types, in this work we focus on the key invariants which are captured by abstract refinements.

- `Micro`, which includes several functions demonstrating parametric reasoning with base values, type classes, and higher-order loop invariants for traversals and folds, as described in § 2.1 and § 2.4;
- `Vector`, which includes the domain- and range-generic `Vec` functions and several "clients" that use the generic `Vec` to implement incremental initialization, null-terminated strings, and memoization, as described in § 2.2;
- `ListSort`, which includes various textbook sorting algorithms including insert-, merge- and quick-sort. We verify that the functions actually produce sorted lists, *i.e.,* are of type `IncrList a`, as described in § 2.3;
- `Data.List.sort`, which includes three non-standard, optimized list sorting algorithms, as found in the `base` package. These employ lists that are increasing and decreasing, as well as lists of (sorted) lists, but we can verify that they also finally produce values of type `IncrList a`;
- `Data.Set.Splay`, which is a purely functional, top-down splay set library from the `llrbtree` package. We verify that all the interface functions take and return binary search trees;
- `Data.Map.Base`, which is the widely-used implementation of functional maps from the `containers` package. We verify that all the interface functions preserve the crucial binary search ordering property and various related invariants.

Table 1 quantitatively summarizes the results of our evaluation. We now give a qualitative account of our experience using HSOLVE by discussing what the specifications and other annotations look like.

Specifications are usually simple. In our experience, abstract refinements greatly simplify writing specifications for the *majority* of interface or public functions. For example, for `Data.Map.Base`, we defined the refined version of the `Tree` ADT (actually called `Map` in the source, we reuse the type from § 2.3 for brevity), and then instantiated it with the concrete refinements for binary-search ordering with the alias `BST k v` as described in § 2.3. Most refined specifications were just the Haskell types with the `Tree` type constructor replaced with the alias `BST`. For example, the type of `fromList` is refined from `(Ord k)=> [(k, a)] -> Tree k a` to `(Ord k)=> [(k, a)] -> BST k a`. Furthermore, intra-module Liquid type inference permits the automatic synthesis of necessary stronger types for private functions.

Auxiliary Invariants are sometimes Difficult. However, there are often rather thorny *internal* functions with tricky invariants, whose specification can take a bit of work. For example, the function `trim` in `Data.Map.Base` has the following behavior (copied verbatim from the documentation): "`trim blo bhi t` trims away all subtrees that surely contain no values between the range `blo` to `bhi`. The returned tree is either empty or the key of the root is between `blo` and `bhi`." Furthermore `blo` (resp. `bhi`) are specified as option (*i.e.,* `Maybe`) values with **`Nothing`** denoting $-\infty$ (resp. $+\infty$). Fortunately, refinements suffice to encode such properties. First, we define measures

```
measure isJust      :: Maybe a -> Bool
isJust  (Just x)     = true
isJust  (Nothing)    = false

measure fromJust    :: Maybe a -> a
fromJustS  (Just x)  = x

measure isBin         :: Tree k v -> Bool
isBin  (Bin _ _ _ _)  = true
isBin  (Tip)          = false

measure key :: Tree k v -> k
key  (Bin k _ _ _)    = k
```

which respectively embed the `Maybe` and `Tree` root value into the refinement logic, after which we can type the `trim` function as

```
trim :: (Ord k) => blo:Maybe k
                -> bhi:Maybe k
                -> BST k a
                -> {v:BST k a | bound(blo, v, bhi)}
```

where `bound` is simply a refinement alias

```
refinement bound(lo, v, hi)
  =  isBin(v) => isJust(lo) => fromJust(lo) < key(v)
  && isBin(v) => isJust(hi) => fromJust(hi) > key(v)
```

That is, the output refinement states that the root is appropriately lower- and upper-bounded if the relevant terms are defined. Thus, refinement types allow one to formalize the crucial behavior as machine-checkable documentation.

Code Modifications. On a few occasions we also have to change the code slightly, typically to make explicit values on which various invariants depend. Often, this is for a trivial reason; a simple re-ordering of binders so that refinements for *later* binders can depend on earlier ones. Sometimes we need to introduce "ghost" values so we can write the specifications (*e.g.,* the `foldr` in § 2.4). Another example is illustrated by the use of list `append` in `quickSort`. Here, the `append` only produces a sorted list if the two input lists are sorted and such that each element in the first is less than each element in the second. We address this with a special `append` parameterized on `pivot`

```
append :: pivot:a
       -> IncrList {v:a | v < pivot}
       -> IncrList {v:a | v > pivot}
       -> IncrList a
append pivot [] ys     = pivot : ys
append pivot (x:xs) ys = x : append pivot xs ys
```

5 Related Work

The notion of type refinements was introduced by Freeman and Pfenning [11], with refinements limited to restrictions on the structure of algebraic datatypes, for which inference is decidable. Our present notion of refinement types has its roots in the *indexed types* of Xi and Pfenning [26], wherein data types' ranges are restricted by *indices*, analogous to our refinement predicates, drawn from a decidable domain; in the example case explored by Xi and Pfenning, types were indexed by terms from Presburger arithmetic. Since then, several approaches to developing richer refinement type systems and accompanying methods for type checking have been developed. Knowles and Flanagan [17] allow refinement predicates to be arbitrary terms of the language being typechecked and present a technique for deciding some typing obligations statically and deferring others to runtime. Findler and Felleisen's [8] higher-order contracts, which extend Eiffel's [18] first-order contracts — ordinary program predicates acting as dynamic pre- and post-conditions — to the setting of higher-order programs, eschew any form of static checking, and can be seen as a dynamically-checked refinement type system. Bengtson et al. [2] present a refinement type system in which type refinements are drawn from a decidable logic, making static type checking tractable. Greenberg et al. [1] gives a rigorous treatment of the metatheoretic properties of such a refinement type system.

Refinement types have been applied to the verification of a variety of program properties [26, 7, 2, 10]. In the most closely related work to our own, Kawaguchi et al. [16] introduce *recursive* and *polymorphic* refinements for data structure properties. The present work unifies and generalizes these two somewhat ad-hoc notions into a single, strictly and significantly more expressive mechanism of abstract refinements.

A number of higher-order logics and corresponding verification tools have been developed for reasoning about programs. Example of systems of this type include NuPRL [4], Coq [3], $F^\star$ [24] and Agda [20] which support the development and verification of higher-order, pure functional programs. While these systems are highly expressive, their expressiveness comes at the cost of making logical validity checking undecidable. To help automate validity checking, both built-in and user-provided tactics are used to attempt to discharge proof obligations; however, the user is ultimately responsible for manually proving any obligations which the tactics are unable to discharge.

References

1. J. F. Belo, M. Greenberg, A. Igarashi, and B. C. Pierce. Polymorphic contracts. In *ESOP*, pages 18–37, 2011.
2. J. Bengtson, K. Bhargavan, C. Fournet, A. D. Gordon, and S. Maffeis. Refinement types for secure implementations. *ACM TOPLAS*, 33(2):8, 2011.
3. 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.
4. R.L. Constable. *Implementing Mathematics with the Nuprl Proof Development System*. Prentice-Hall, 1986.
5. P. Cousot, R. Cousot, and F. Logozzo. A parametric segmentation functor for fully automatic and scalable array content analysis. In *POPL*, pages 105–118, 2011.
6. L. de Moura and N. Bjørner. Z3: An efficient SMT solver. In *TACAS*, 2008.
7. J. Dunfield. *A Unified System of Type Refinements*. PhD thesis, Carnegie Mellon University, Pittsburgh, PA, USA, 2007.
8. R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *ICFP*, pages 48–59, 2002.
9. C. Flanagan, R. Joshi, and K. R. M. Leino. Annotation inference for modular checkers. *Information Processing Letters*, 2001.
10. C. Fournet, M. Kohlweiss, and P-Y. Strub. Modular code-based cryptographic verification. In *CCS*, pages 341–350, 2011.
11. T. Freeman and F. Pfenning. Refinement types for ML. In *PLDI*, 1991.
12. D. Gopan, T. W. Reps, and S. Sagiv. A framework for numeric analysis of array operations. In *POPL*, pages 338–350, 2005.
13. S. Graf and H. Saïdi. Construction of abstract state graphs with PVS. In *CAV*, LNCS 1254, pages 72–83. Springer, 1997.
14. A. Guha, M. Fredrikson, B. Livshits, and N. Swamy. Verified security for browser extensions. In *IEEE Symposium on Security and Privacy*, pages 115–130, 2011.
15. R. Jhala and K. L. McMillan. Array abstractions from proofs. In *CAV*, pages 193–206, 2007.
16. M. Kawaguchi, P. Rondon, , and R. Jhala. Type-based data structure verification. In *PLDI*, pages 304–315, 2009.
17. K.W. Knowles and C. Flanagan. Hybrid type checking. *ACM TOPLAS*, 32(2), 2010.
18. B. Meyer. *Eiffel: the language*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.
19. G. Nelson. Techniques for program verification. Technical Report CSL81-10, Xerox Palo Alto Research Center, 1981.
20. U. Norell. *Towards a practical programming language based on dependent type theory*. PhD thesis, Chalmers, SE-412 96 Göteborg, Sweden, September 2007.
21. X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP TCS*, pages 437–450, 2004.
22. P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.
23. T. Sheard. Type-level computation using narrowing in omega. In *PLPV*, 2006.
24. N. Swamy, J. Chen, C. Fournet, P-Y. Strub, K. Bhargavan, and J. Yang. Secure distributed programming with value-dependent types. In *ICFP*, pages 266–278, 2011.
25. D. Vytiniotis, S. L. Peyton Jones, and J. Magalhães. Equality proofs and deferred type errors: a compiler pearl. In *ICFP*, pages 341–352, 2012.
26. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.