Embedded Domain Specific Verifiers

Ranjit Jhala

University of California, San Diego

Abstract. Refinement types allow for automatic, SMT-based compositional specification and verification of program properties by liberating assertions from control locations. Recent work has shown how to increase the expressiveness of specifications by allowing arbitrary user-defined functions within assertions, while still preserving decidable verification. In this paper we illustrate the benefits of this expressiveness by showing how to embed a verified Floyd-Hoare style verifier *within* the refinement type checker LIQUIDHASKELL, *i.e.,* we show how to implement a library function `verify p c q` that only automatically type checks only when the Floyd-Hoare triple {p} c {q} is legitimate.

1 Introduction

Refinement types are a generalization of Floyd-Hoare style contracts where *logical assertions* are liberated from *control* elements (*e.g.,* function or loop entries) and instead associated directly the *data* on which the code operates. For example, the refinement type `{v:Int|0 < v}` denotes the basic type `Int` refined with a logical assertion that restricts the underlying values to be positive. Similarly, the (dependent) function type `x:{Int|0 < x}` $\rightarrow$ `{v:Int|v < x}` describes functions that take a positive argument `x` and return outputs exceeding `x`.

Refinements allow the programmer to specify complex properties by *composing* quantifier-free predicates with type constructors, *e.g.,* writing `List {v:Int|0 < v}` to describe a list of positive integers. Dually, they allow the machine to automatically verify those properties by *decomposing* the corresponding subsumption checks, *e.g.,* to verify that the `List {v:Int|a` $\leq$ `v}` is subsumed by the list of positives when `a` is positive, by asking an SMT solver to validate the formula $0 < a \Rightarrow a \leq v \Rightarrow 0 < v$.

Earlier work focused on relatively simpler properties like array bounds checking [33, 24], data structure invariants [9] or security policies [4] by restricting the language of assertions to linear arithmetic and uninterpreted functions which yield SMT-decidable validity queries. However, recent work has unearthed several ways to encode sophisticated specifications within the contracts, *e.g.,* using Horn clauses [28] or user-defined functions, thereby permitting "computation" within the assertions [1, 30]. This added expressiveness makes it possible to encode a variety of *domain-specific* constraints within the type system, effectively implementing new kinds of code verification simply as type-checking with a suitably refined interface. For example, [23] shows how to encode Information Flow Control using refinements, and [19] shows how to then encode the semantics of SQL operators within refinement types in order to precisely and automatically track security policies across web applications.

In this paper we illustrate this method of *embedded domain specific verification* by using the refinement type based verifier LIQUIDHASKELL [29], to implement a verified Floyd-Hoare style verifier [10, 13] for a small imperative language IMP. Crucially, the verification of IMP code itself is carried out via LIQUIDHASKELL's automatic refinement type checking. That is, we show how to implement a (library) function `verify` such that `verify p c q` only type checks for legitimate Floyd-Hoare triples $\{p\}\ c\ \{q\}$. We develop `verify` via the following concrete steps.

1. Programs and Operational Semantics. First, in § 3, we show how to represent the syntax of IMP *programs* via plain datatypes, and illustrate how to represent their "big-step" semantics via refined datatypes that represent valid executions.

2. Axiomatic Semantics. Second, in § 4, we recall the elements of Floyd-Hoare *logic* and show how proofs in the program logic can also be reified as refined data. We use this data to formally verify the soundness of the program logic by writing a function that uses the structure reifying each proof to demonstrate the legitimacy of the triple.

3. Verification Conditions. Third, in § 5, we show how to turn the deductive Floyd-Hoare proof system into an *algorithm* via a function `vc p c q` that converts each triple into a *verification condition*: a logical formula whose validity implies the legitimacy of the triple. We establish the soundness of `vc` generation by using the validity of `vc p c q` to construct a Floyd-Hoare derivation for the triple $\{p\}\ c\ \{q\}$.

4. An Embedded Verifier. Finally, in § 5.2 we use `vc` to implement `verify` as a function whose precondition requires that `vc p c q` be valid. That is, expressions `verify p c q` are well typed only when the corresponding VC is valid, and further the refinement typing reduces exactly to checking the validity of the computed VC. We conclude by showing how to use our embedded verifier to check some IMP programs via refinement typing. [1]

2 Refinement Types

Lets begin with a brisk introduction to refinement types that illustrates how they can be used to formally specify and verify properties of programs.

2.1 Refinements

A *refinement type* a plain type like `Int` or `Bool` decorated by a logical predicate which restricts the *set of values*. For example, the refinement type `Nat` defined as

```
type Nat = {v:Int|0 ≤ v}
```

denotes the set of non-negative `Int`eger values.

Specification. Consider the function `sum n` which computes $1 + \ldots + n$

```
sum :: n:Nat → {v:Int|n ≤ v}
sum n = if n == 0 then 0 else n + sum (n - 1)
```

[1] This entire paper is written as a program verified by LIQUIDHASKELL that the interested reader may find at https://github.com/liquidhaskell/floyd-hoare.

The signature for `sum` is a refined *function* type that specifies: (1) A *pre-condition* that `sum` only be called with non-negative inputs `n`, (as otherwise it diverges), and (2) A *post-condition* that states that the output produced by the function is no smaller than `n`.

Under classical *strict* or *call-by-value* evaluation semantics (where a function's arguments must be evaluated before the call proceeds) refinement types represent *safety* properties, *i.e.,* they correspond to the classic notion of *partial correctness*. That is, the output type only specifies that *if* a value is returned, it will exceed the input `n`. *Lazy* evaluation muddies the waters introducing an unexpected intertwining of safety and termination [29]. For simplicity, we will assume that that the underlying language follows the usual strict semantics, and not concern ourselves with termination.

Verification. Refinement type checking algorithmically verifies the implementation of `sum` meets its specification by (1) computing a logical formula called a *verification condition* (VC) (2) and then using an SMT solver to check the *validity* of the VC [14]. For `sum` the VC is the formula that combines the assumption $0 \leq n$ from the precondition to establish that the returned values $v = 0$ and $v = n + sum(n-1)$ in each branch are indeed greater than n. In the latter case, the assumption $n - 1 \leq sum(n-1)$ is established by "inductively" assuming the post-condition of `sum` for the recursive call.

$$\forall n, v.\ 0 \leq n \Rightarrow n = 0 \Rightarrow v = 0 \Rightarrow n \leq v \wedge \\ n \neq 0 \Rightarrow n - 1 \leq sum\ (n-1) \Rightarrow v = n + sum(n-1) \Rightarrow n \leq v$$

Assertions. We can encode *assertions* as function calls by defining a function that requires its input be a `Bool` that is `True`

```
assert :: {b:Bool|b} → ()
assert b = ()
```

Subsequent refinement type checking then statically verifies classical assertions

```
checkSum1 = assert (1 ≤ sum 1)
```

2.2 Reflection

Refinement type checking is *modular*, which means that at call-sites, the only information known about a function is *shallow*, namely that which is stated in its type specification. Consequently, the following assertion fails to verify

```
checkSum2 = assert (3 ≤ sum 2)
```

The only information about the call `sum 2` is that encoded in its type specification: that the output exceeds 2, and so `checkSum2` fails to verify due the *invalid* VC

$$2 \leq sum(2) \Rightarrow 3 \leq sum(2)$$

Reflecting Implementations into Specifications. To prove *deeper* properties about `sum` we can reflect the implementation (*i.e.,* definition) of the function into its specification. To do so, the programmer writes `{-@ reflect sum @-}` which *strengthens* the specification of `sum` to n`:Nat` $\rightarrow$ `{`v`:`$n \leq v \wedge \phi_{sum}(n, v)$`}` where

$$\phi_{sum}(n, v) \doteq (v = sum(n)) \wedge (n = 0 \Rightarrow v = 0) \wedge (n \neq 0 \Rightarrow v = n + sum(n-1))$$

Logical Evaluation. To ensure that validity checking remains *decidable*, sum is *uninterpreted* in the refinement logic. This means, that the new VC for `checkSum`

$$\forall b.\ 2 \leq sum(2) \wedge \phi_{sum}(2, sum(2)) \Rightarrow b \Leftrightarrow (3 \leq sum(2)) \Rightarrow b$$

is still *invalid* as there is no information about $sum(1)$. Fortunately, the method of *Proof by Logical Evaluation* (PLE) [30] lets the SMT solver strengthen the hypotheses by *unfolding the definition* of `sum` to yield the following valid VC that verifies `checkSum2`.

$$\phi_{sum}(0, sum(0)) \wedge \phi_{sum}(1, sum(1)) \wedge 2 \leq sum(2) \wedge \phi_{sum}(2, sum(2)) \Rightarrow 3 \leq sum(2)$$

Proofs by Induction. Reflection and logical evaluation let us specify and verify more interesting properties about `sum`. For example, consider the following *tail-recursive* version which can be compiled into an efficient loop

```
reflect sumTR'
sumTR' :: Nat → Int → Int
sumTR' n a = if n == 0 then a else sumTR' (n-1) (a+n)

reflect sumTR
sumTR :: Nat → Int
sumTR n = sumTR' n 0
```

We can now specify that `sumTR` is *equivalent* to `sum` by first establishing that

$$\forall n, a.\ 0 \leq n \ \Rightarrow\ \mathsf{sumTR'}(n, a) = a + \mathsf{sum}(n) \tag{1}$$

and then using the lemma to prove that

$$\forall n.\ 0 \leq n \ \Rightarrow\ \mathsf{sumTR}(n) = \mathsf{sum}(n) \tag{2}$$

Proofs as Programs. Our approach uses the Curry-Howard correspondence [32] to encode logical *propositions* as refinement types, and provide *proofs* of the propositions by writing functions of the appropriate type. Briefly, Curry-Howard shows how universally quantified propositions $\forall x : \texttt{Int}.\ P(x)$ correspond to the function type `x:Int →{P(x)}`, and that code implementing the above type is a constructive proof of the original proposition. For example, the proposition eq. (1) is specified and verified by

```
lem_sumTR :: n:Nat → acc:Int → {sumTR' n acc = acc + sum n}
lem_sumTR 0 _   = ()
lem_sumTR n acc = lem_sumTR (n - 1) (acc + n)
```

At a high-level, the the method of logical evaluation is able to verify the above code by using the post-condition from the recursive invocation of `lem_sumTR` (that "adds" the induction hypothesis as an antecedent of the VC) and automatically unfolding the definitions of `sumTR'` and `sum`. Similarly, we translate eq. (2) and prove that it is a corollary of eq. (1) by "calling" the lemma (function) with the appropriate arguments.

```
thm_sumTR :: n:Nat → { sumTR n == sum n }
thm_sumTR n = lem_sumTR n 0
```

2.3 Reification

Some proofs require the ability to *introspect on the evidence* that establishes why some proposition holds. As a textbook example, let us recall the notion of the *reachability* in a graph. Let V and $E \subset V \times V$ denote the set of *vertices* and *directed edges* of a directed graph. We say that a vertex u *reaches* v if either (a) $u = v$ or (b) $(u, v) \in E$ and v reaches w. Suppose that we wish to prove that the notion of reachability is *transitive*, *i.e.,* if u reaches v and v reaches w then u reaches w. To prove the above property, it is not enough to know *that* u reaches v and v reaches w. Instead, we need additional evidence: a "path" that describes *how* the evidence was established.

Our third key piece of machinery is a way to *reify* such evidence using data types that can be introspected and manipulated to provide evidence that establishes new properties. This machinery corresponds to the notion of *inductive* propositions or predicates used by proof assistants like Coq or Isabelle [5, 22]. As an example, lets see how to formalize the notion of reachability. Suppose that we represent the directed edge relation as a predicate over vertices `v`

```
type Edge v = v → v → Bool
```

Step 1: Propositions as Data. We encode reachability as a relation $x \longrightarrow_e^* y$ that says x reaches y following the edges e. We can represent this relation as proposition: a *value* `Reach e u v` that denotes that `u` reaches `v` following the edges `e`.

Step 2: Evidence as Data. We can specify reachability via two rules

$$\frac{}{x \longrightarrow_e^* x}[\textsc{Self}] \qquad \frac{(x,y) \in e \qquad y \longrightarrow_e^* z}{x \longrightarrow_e^* z}[\textsc{Step}]$$

We can formally represent the *evidence* of reachability as a refined datatype whose constructors correspond to the informal rules.

```
data ReachEv a where
       Self :: e:Edge a → x:a →
               Reach e x x
       Step :: e:Edge a → x:a → y:{a|e x y} → z:a →
               Reach e y z →
               Reach e x z
```

Following the Curry-Howard correspondence, (1) the universally quantified variables of the rules become input *parameters* for the constructor, (2) the antecedents of each rule are translated to *preconditions* for the corresponding constructor, and (3) the consequent of each rule is translated into the *postcondition* for the constructor. The above datatype says there are exactly two ways to *construct* evidence of reachability: (1) `Self e x` is evidence that a vertex `x` can reach *itself* following the edge-relation `e`; (2) `Step e x y z yz` uses the fact that (a) `x` has an edge to `y` (established by the precondition `e x y`) and (b) `y` reaches `z` (established by `yz` which is evidence that `Reach e y z`) to construct evidence that `x` reaches `z`. Note that the above are the *only* ways to provide evidence of reachability (*i.e.,* to construct values that demonstrate the proposition `Reach e x y`).

Step 3: Consuming and Producing Evidence. Finally, we can prove properties about reachability, simply by writing functions that consume and produce evidence. For example, here is a proof that reachability is transitive.

```
reachTrans :: e:Edge a → x:a → y:a → z:a →
              Reach e x y → Reach e y z →
              Reach e x z
reachTrans e x y z (Self _ _) yz          = yz
reachTrans e x y z (Step _ _ x1 _ x1y) yz = Step e x x1 z x1z
  where x1z = reachTrans e x1 y z x1y yz
```

The *specification* of `reachTrans` represents the proposition that reachability is transitive: for any edge relation `e` and vertices `x,y,z` if we have evidence that `x` reaches `y` and that `y` reaches `z` then we can construct evidence that `x` reaches `z`. The *implementation* of `reachTrans` demonstrates the claim via code that explicitly constructs the evidence via recursion, *i.e.,* by induction on the path from `x` to `y`. In the base case, that path is empty as `x` equals `y`, in which case the evidence `yz` that shows `y` reaches `z` *also* shows `x` reaches `z`. In the inductive case, the path from `x` to `y` goes via the edge from `x` to `x1` followed by a path `x1y` from `x1` to `y`. (As when proving `thm_sumTR`) we *apply* the induction hypothesis by recursively "calling" `reachTrans` on the sub-paths `x1y` and `yz` to obtain evidence that `x1` reaches `z`, and then link that evidence the edge from `x` to `x1` (*i.e.,* `Step e x x1`) to construct the path from `x` to `z`. (The termination checker automatically verifies that the recursion in `reachTrans`, and hence the induction, is well-founded [30].)

3 Programs

Next, lets spell out the syntax and semantics of IMP, that language that we wish to build a verifier for. We will define a datatype to represent the *syntax* of IMP programs and then formalize their *semantics* by defining evaluation functions and transition rules.

3.1 Syntax

IMP is a standard imperative language with integer valued variables, arithmetic expressions, boolean conditions, assignments, branches and loops.

Variables and Expressions. IMP has (integer valued) identifiers `Id` and arithmetic expressions `AExp`, represented by the datatypes

```
type Val = Int
type Id = String

data AExp
  = N Val           -- ^ 0,1,2...
  | V Id            -- ^ x,y,z...
  | Plus  AExp AExp -- ^ e1 + e2
  | Minus AExp AExp -- ^ e1 - e2
  | Times AExp AExp -- ^ e1 * e2
```

We can define infix operators

```
b1 .+. b2 = Plus  b1 b2
b1 .-. b2 = Minus b1 b2
b1 .*. b2 = Times b1 b2
```

For example, we can represent the expression $x + 2 * y$ as

```
e0 = V "x" .+. (N 2 .*. V "y")
```

Conditions. We use relations on integer expressions to build *conditions* which can be further combined using boolean operators

```
data BExp
  = Bc  Bool        -- ^ True, False
  | Not BExp        -- ^ not b
  | And BExp BExp   -- ^ b1 && b2
  | Leq AExp AExp   -- ^ a1 ≤ a2
  | Eql AExp AExp   -- ^ a1 == a2
```

We can define other relations and boolean operations using the above.

```
e1 .==. e2 = Eql e1 e2
e1 .!=. e2 = Not (e1 .==. e2)
b1 .≤.  b2 = Leq b1 b2
b1 .<.  b2 = (b1 .≤. b2) .&&. (b1 .!=. b2)
b1 .&&. b2 = And b1 b2
b1 .||. b2 = Not (Not b1 .&&. Not b2)
b1 .=>. b2 = (Not b1) .||. b2
```

Commands. We can use `AExp` and `BExp` to define the syntax of *commands* `Com`

```
data Com
  = Skip                      -- skip
  | Assign Id    AExp         -- x ← a
  | Seq    Com   Com          -- c1; c2
  | If     BExp  Com   Com    -- if b then c1 else c2
  | While  BExp  BExp  Com    -- while {inv} b c
```

We can introduce some helper functions to improve readability, *e.g.,*

```
(←) :: Id → AExp → Com
x ← e = Assign x e
```

The following IMP program sums up the integers from `1` to `n` with the result stored in `r`

```
com_sum inv =
  ("i" ← N 0) @@                  -- i ← 0;
  ("r" ← N 0) @@                  -- r ← 0;
  While inv (V "i" .!=. V "n")    -- WHILE (i != n)
    ("r" ← V "r" .+. V "i") @@    --  r ← r + i;
    ("i" ← V "i" .+. N 1)         --  i ← i + 1
```

3.2 Semantics

Next, we the semantics of programs via *states*, *valuations* and *transitions*. A *state* is map from `Id`entifiers to (integer) `Val`ues

```
type State = Map Id Val
```

where a `Map k v` is a sequence of key-value pairs, with a `Def`ault value for missing keys

```
data Map k v = Def v | Set k v (Map k v)
```

We can `set` the value of a variable and `get` the value of a variable in a state, via

```
get :: (Eq k) => Map k v → k → v
get (Def v)      _  = v
get (Key k v s) k' = if k == k' then v else get s k'

set :: Map k v → k → v → Map k v
set s k v = Set k v s
```

For example, suppose that `s0` is the `State`

```
s0 = set "y" 30 (set "x" 20 (set "y" 10 (def 0)))
```

Then `get "x" s0` and `get "y" s0` respectively evaluate to `20` and `30` and `get z s0` evaluates to `0` for any other identifer `z`.

Evaluating Expressions and Conditions. We can lift the notion of valuations from states to arithmetic expressions via the function `aval`

```
aval                  :: AExp → State → Val
aval (N n) _          = n
aval (V x) s          = get s x
aval (Plus  e1 e2) s = aval e1 s + aval e2 s
aval (Minus e1 e2) s = aval e1 s - aval e2 s
aval (Times e1 e2) s = aval e1 s * aval e2 s
```

For example `aval e0 s0` evaluates to `80` where `s0` is the state where `"x"` and `"y"` were respectively `"20"` and `"30"` and `e0` is the expression representing $x + 2 \times y$.

Evaluating Conditions. Similarly, we extend the notion of valuations to boolean conditions via the function `bval`

```
bval :: BExp → State → Bool
bval (Bc   b)      _ = b
bval (Not  b)      s = not (bval b s)
bval (And  b1 b2) s = bval b1 s && bval b2 s
bval (Leq  a1 a2) s = aval a1 s ≤ aval a2 s
bval (Eql a1 a2)  s = aval a1 s == aval a2 s
```

For example, is `x_lt_y = V "x" .<. V "y"` then `bval x_lt_y s0` is `True`.

Transitions. The execution of a `Com`mand `c` from a state `s` *transitions* the system to some successor state `s'`. The direct route would be to formalize transitions as a function that takes as input a command and input state `s` and returns the successor `s'` as output. Unfortunately, this path is blocked by two hurdles. First, the function is *partial* as for

Big-Step Transition $\boxed{\langle c, s\rangle \Downarrow s'}$

$$\frac{}{\langle \texttt{Skip}, s\rangle \Downarrow s}\,[\text{B-SKIP}] \qquad \frac{}{\langle x \leftarrow e, s\rangle \Downarrow \texttt{set}\ x\ \texttt{aval}\ e\ s}\,[\text{B-ASSIGN}]$$

$$\frac{\langle c_1, s\rangle \Downarrow s' \qquad \langle c_2, s'\rangle \Downarrow s''}{\langle \texttt{Seq}\ c_1\ c_2, s\rangle \Downarrow s''}\,[\text{B-SEQ}] \qquad \frac{\texttt{bval}\ b\ s \qquad \langle c_1, s\rangle \Downarrow s'}{\langle \texttt{If}\ b\ c_1\ c_2, s\rangle \Downarrow s'}\,[\text{B-IF-T}]$$

$$\frac{\neg\texttt{bval}\ b\ s \qquad \langle c_2, s\rangle \Downarrow s'}{\langle \texttt{If}\ b\ c_1\ c_2, s\rangle \Downarrow s'}\,[\text{B-IF-F}] \qquad \frac{\neg\texttt{bval}\ b\ s}{\langle \texttt{While}_I\ b\ c, s\rangle \Downarrow s}\,[\text{B-WHILE-F}]$$

$$\frac{\texttt{bval}\ b\ s \qquad \langle c, s\rangle \Downarrow s' \qquad \langle \texttt{While}_I\ b\ c, s'\rangle \Downarrow s''}{\langle \texttt{While}_I\ b\ c, s\rangle \Downarrow s''}\,[\text{B-WHILE-T}]$$

Fig. 1. Big-Step Transitions for IMP commands.

certain starting states `s`, certain commands `c` may be *non-terminating*. Second, more importantly, our Floyd-Hoare soundness proof will require a form of induction on the execution traces that provide evidence of *how* `s` transitioned to `s'`.

Big-Step Semantics. Thus, we represent the transitions via the classical *big-step* (or *natural*) style where $\langle c, s\rangle \Downarrow s'$ indicates that the execution of command c transitions the machine from a state s to s'. The rules in fig. 1 characterize the transitions in terms of sub-commands of c. [B-SKIP] states that `Skip` leaves the state unchanged (*i.e.,* yields a transition from s to s). [B-ASSIGN] says that the command `Assign x e` transitions the system from s to a new state where the of x has been set to `aval` e s: the valuation of e in se [B-SEQ] says that `Seq` c_1 c_2 transitions the system from s to s'' if c_1 transitions s to some s' and c_2 transitions s' to s''. The rules for sequencing branches [B-IF-T, B-IF-F] and loops [B-WHL-T, B-WHL-F] similarly describe how the execution of the sub-commands yield transitions from the respective input states to their outputs, using `bval` to select the appropriate sub-command for conditionals.

Reifying Transitions as a Refined Datatype. We represent the big-step transition relation $\langle c, s\rangle \Downarrow s'$ as a *proposition* `BStep c s s'`, and reify the *evidence* that establishes the transitions via the refined datatype in fig. 2. Each rule in fig. 1 is formalized by a data constructor of the corresponding name, whose *input* preconditions mirror the hypotheses of the rule, and whose *output* establishes the postcondition. For example the `BSeq` constructor takes as input the sub-commands `c1` and `c2`, states `s`, `s'` and `s''`, and evidence `BStep c1 s s'` and `BStep c2 s' s''` (that `c1` and `c2` respectively transition `s` to `s'` and `s'` to `s''`) to output evidence that `Seq c1 c2` transitions `s` to `s''`.

This reification addresses both the hurdles that block a direct encoding via a transition function. First, the evidence route sidesteps the problem of non-termination by letting us work with *derivation trees* that correspond exactly to terminating executions. Second, the trees provide a concrete object that describes *how* a state `s` transitioned to `s'` and now we can do inductive proofs over traces, via recursive functions on the derivation trees.

```
data BStep where
     BSkip ::
      s:_ → BStep Skip s s

     BAsgn ::
      x:_ → a:_ → s:_ →
      BStep (Assign x a) s (asgn x a s)

     BSeq  ::
      c1:_ → c2:_ → s:_ → s':_ → s'':_ →
      BStep c1 s s' → BStep c2 s' s'' →
      BStep (Seq c1 c2) s s''

     BIfT  ::
      b:_ → c1:_ → c2:_ → s:{bval b s} → s':_ →
      BStep c1 s s' →
      BStep (If b c1 c2) s s'

     BIfF  ::
      b:_ → c1:_ → c2:_ → s:{not (bval b s)} → s':_ →
      BStep c2 s s' →
      BStep (If b c1 c2) s s'

     BWhlF ::
      i:_ → b:_ → c:_ → s:{not (bval b s)} →
      BStep (While i b c) s s

     BWhlT ::
      i:_ → b:_ → c:_ → s:{bval b s} → s':_ → s'':_ →
      BStep c s s' → BStep (While i b c) s' s'' →
      BStep (While i b c) s s''
```

Fig. 2. Reifying the derivation of $\langle c, s\rangle \Downarrow s'$ with the refined datatype `BStep c s s'`.

4 Deductive Verification

Next, let us build (and verify!) a *deductive* verifier for IMP using the classical method of Floyd-Hoare (FH) logic [10, 13] and show how to prove it sound.

4.1 Floyd-Hoare Triples

Assertions. An *assertion* is a boolean condition over the program identifiers.

```
type Assertion = BExp
```

An assertion `p` *holds* at a state `s` if `bval p s` is `True`, *i.e.,* the assertion evaluates to `True` at the state. For example, the assertion `b0 = V "x".<. V "y"` holds at `s0` where `get s0 "x"` and `get s0 "y"` were respectively `20` and `30`.

Validity. An assertion `p` is *valid* if it holds *for all* states, *i.e.,* $\forall s.$`bval p` s = `True`. Following the Curry-Howard correspondence, we can formalize validity as

```
type Valid P = s:State → {bval P s}
```

Logical evalation [30] makes it easy to check validity simply by refinement typing. For example, we can establish the assertion

```
cond_x_10_5 = (N 10 .≤. V "x") .=>. (N 5 .≤. V "x")
```

is valid, simply by writing

```
pf_valid :: Valid cond_x_10_5
pf_valid = \_ → ()
```

Logical evaluation discharges the typing obligation via the SMT validated VC

$$\forall s.\ 10 \leq \texttt{get}\ \text{“}x\text{”}\ s \Rightarrow 5 \leq \texttt{get}\ \text{“}x\text{”}\ s$$

Floyd-Hoare Triples. A *Floyd-Hoare triple* $\{p\}\ c\ \{q\}$ comprising a *precondition* p, command c and *post-condition* q. The triple states that if the command c transitions the system *from* a state s where the precondition p holds, *to* a state s', then the postcondition q holds on s'.

Legitimacy of Triples. We say a triple is *legitimate*, written $\models \{p\}\ c\ \{q\}$ if

$$\models \{p\}\ c\ \{q\}\ \doteq\ \forall s, s'.\ \texttt{bval}\ p\ s \Rightarrow \langle c, s\rangle \Downarrow s' \Rightarrow \texttt{bval}\ q\ s'$$

We can use the Curry-Howard correspondence to formalize the above notion as:

```
type Legit P C Q =
      s:{bval P s} → s':_ → BStep C s s' → {bval Q s'}
```

We can specify and verify the triple $\{0 \leq x\}\ y \leftarrow x + 1\ \{1 \leq y\}$ as

```
y_x_1 :: Com
y_x_1 = ("y" ← V "x" .+. N 1)

leg_y_x_1 :: Legit (N 0 .≤. V {"x"}) y_x_1 (N 1 .≤. V {"y"})
leg_y_x_1 :: Legit
leg_y_x_1 s s' BAsgn {} = ()
```

Note that evidence for the big-step transition `BAsgn {}` tells us that the final state `s'` is obtained by `set`ting the value of `y` to `1 +` the value of `x` in the initial state `s`. Thus, refinement typing verifies legitimacy by generating the following VC for `leg_y_x_1` (simplified after logical evaluation unfolds the definition of `bval` for the pre- and post-conditions)

$$\forall s, s'.\ 0 \leq \texttt{get}\ \text{“}x\text{”}\ s\ \Rightarrow\ s' = \texttt{set}\ \text{“}y\text{”}\ 1 + \texttt{get}\ \text{“}x\text{”}\ s\ s\ \Rightarrow\ 1 \leq \texttt{get}\ \text{“}y\text{”}\ s'$$

PLE then further unfolds the definition of `set` to allow the SMT solver to automatically verify the VC and hence, check legitimacy.

As a second example, let `x20_y30` be a command that sequentially assigns `x` and `y` to `20` and `30` respectively:

```
bsub :: Id → AExp → BExp → BExp
bsub x a (Bc  b)     = Bc  b
bsub x a (Not b)     = Not (bsub x a b)
bsub x a (And b1 b2) = And (bsub x a b1) (bsub x a b2)
bsub x a (Leq a1 a2) = Leq (sub  x a a1) (sub  x a a2)
bsub x a (Eql a1 a2) = Eql (sub  x a a1) (sub  x a a2)

sub :: Id → AExp → AExp → AExp
sub x e (Plus  a1 a2)  = Plus  (sub x e a1) (sub x e a2)
sub x e (Minus a1 a2)  = Minus (sub x e a1) (sub x e a2)
sub x e (Times a1 a2)  = Times (sub x e a1) (sub x e a2)
sub x e (V y) | x == y = e
sub _ _ a              = a
```

Fig. 3. Substituting a variable `x` with an expression `e`.

```
x20_y30 = ("x" ← N 20) @@ ("y" ← N 30)
```

We can verify that $\models \{true\}$ `x20_y30` $\{x \leq y\}$ as

```
legXY :: Legit bTrue x20_y30 (V {"x"} .≤. V {"y"})
legXY s s'' (BSeq _ _ _ _ _ (BAsgn {}) (BAsgn {})) = ()
```

Here, the "pattern-matching" on the refined `BStep e` vidence establishes that the final state `s'' = set "y" 30 s'` where the intermediate state `s' = set "x" 20 s`. Thus, refinement typing for `legXY` proceeds by generating the VC

$$\forall s, s', s''.s' = \texttt{set}\ \text{“}x\text{”}\ 20\ s\ \Rightarrow\ s'' = \texttt{set}\ \text{“}y\text{”}\ 30\ s'\ \Rightarrow\ \texttt{get}\ \text{“}x\text{”}\ s'' \leq \texttt{get}\ \text{“}y\text{”}\ s''$$

which is readily validated by the SMT solver.

4.2 Floyd-Hoare Logic

The above examples show that while establishing the legitimacy of triples *explicitly* from the big-step semantics is possible, it quickly gets tedious for complex code. The ingenuity of Floyd and Hoare lay in their design of a recipe to derive triples based on *symbolically* transforming the assertions in a *syntax* directed fashion.

Substitutions. The key transformation is a means to *substitute* all occurrences of an identifier `x` with an expression `e` in a boolean assertion, as formalized respectively by `sub` and `bsub` in fig. 3.

Derivation Rules. We write $\vdash \{p\}\ c\ \{q\}$ to say there exists a tree whose root is the triple denoted by p, c and q, using the syntax-directed rules in fig. 4. As with the big-step semantics we can reify the evidence corresponding to a Floyd-Hoare derivation via the refined datatype `FH p c q` specified in fig. 5. Note that the constructors for the datatype mirror the corresponding derivation rules in fig. 4, and that we have split the classic rule of *consequence* into separate rules for strengthening preconditions ([FH-PRE]) and weakening postconditions ([FH-POST]).

Floyd-Hoare Logic $\boxed{\vdash \{p\}\ c\ \{q\}}$

$$\frac{}{\vdash \{\texttt{Skip}\}\ q\ \{q\}}\text{[FH-SKIP]} \qquad \frac{}{\vdash \{\texttt{bsub}\ x\ e\ q\}\ x \leftarrow e\ \{q\}}\text{[FH-ASGN]}$$

$$\frac{\vdash \{p\}\ c_1\ \{q\} \qquad \vdash \{q\}\ c_2\ \{r\}}{\vdash \{p\}\ \texttt{Seq}\ c_1\ c_2\ \{r\}}\text{[FH-SEQ]}$$

$$\frac{\vdash \{p \wedge b\}\ c_1\ \{q\} \qquad \vdash \{p \wedge \neg b\}\ c_2\ \{q\}}{\vdash \{p\}\ \texttt{If}\ b\ c_1\ c_2\ \{q\}}\text{[FH-IF]}$$

$$\frac{\vdash \{inv \wedge b\}\ c\ \{inv\}}{\vdash \{inv\}\ \texttt{While}\ b\ c\ \{inv \wedge \neg b\}}\text{[FH-WHL]} \qquad \frac{\texttt{Valid}(p' \Rightarrow p) \qquad \vdash \{p\}\ c\ \{q\}}{\vdash \{p'\}\ c\ \{q\}}\text{[FH-PRE]}$$

$$\frac{\vdash \{p\}\ c\ \{q\} \qquad \texttt{Valid}(q \Rightarrow q')}{\vdash \{p\}\ c\ \{q'\}}\text{[FH-POST]}$$

Fig. 4. Syntax-driven derivation rules for Floyd-Hoare Logic

4.3 Soundness

Next, let us verify that the Floyd-Hoare derivation rules are *sound*, *i.e.,* that $\vdash \{p\}\ c\ \{q\}$ implies $\models \{p\}\ c\ \{q\}$. To do so, we will prove *legitimacy* lemmas that verify that if the antecedents of each derivation rule describe legitimate triples, then the consequent is also legitimate.

Legitimacy for Assignments. For assignments we prove, by induction on the structure of the assertion `q`, a lemma that connects state-update with substitution

```
lemBsub :: x:_ → a:_ → q:_ → s:_ →
   {bval (bsub x a q) s = bval q (set x (aval a s) s) }
```

after which we use the big-step definition to verify

```
lemAsgn :: x:_ → a:_ → q:_ →
   Legit (bsub x a q) (Assign x a) q
```

Legitimacy for Branches and Loops. Similarly, for branches and loops we use the big-step derivations to respectively prove that

```
lemIf :: b:_ → c1:_ → c2:_ → p:_ → q:_ →
   Legit (p .&&. b) c1 q → Legit (p .&&. Not b) c2 q →
   Legit p (If b c1 c2)  q
```

The proof (*i.e.,* implementation of `lem_if`) proceeds by splitting cases on the big-step derivation used for the branch and applying the legitimacy function for the appropriate branch to prove the postcondition `q` holds in the output.

```
lemIf b c1 c2 p q l1 l2 =
  \s s' bs → case bs of
    BIfT _ _ _ _ _ c1_s_s' →  -- then branch
```

```
data FH where
     FHSkip :: q:_ →
               FH q Skip q

     FHAsgn :: q:_ → x:_ → a:_ →
               FH (bsub x a q) (Assign x a) q

     FHSeq  :: p:_ → c1:_ → q:_ → c2:_ → r:_ →
               FH p c1 q → FH q c2 r →
               FH p (Seq c1 c2) r

     FHIf :: p:_ → q:_ → b:_ → c1:_ → c2:_ →
             FH (p .&&. b) c1 q → FH (p .&&. Not b) c2 q →
             FH p (If b c1 c2) q

     FHWhl :: inv:_ → b:_ → c:_ →
              FH (inv .&&. b) c inv →
              FH inv (While inv b c) (inv .&&. Not b)

     FHPre  :: p':_ → p:_ → q:_ → c:_ →
               Valid (p' .=>. p) → FH p c q →
               FH p' c q

     FHPost :: p:_ → q:_ → q':_ → c:_ →
               FH p c q → Valid (q .=>. q') →
               FH p c q'
```

Fig. 5. Reifying Floyd-Hoare Proofs as a Refined Datatype

```
        l1 s s' c1_s_s'           -- ... post-cond via l1
      BIfF _ _ _ _ _ c2_s_s' →    -- else branch
        l2 s s' c2_s_s'           -- ... post-cond via l2
```

The legitimacy lemma for loops is similar.

```
lemWhile :: b:_ → c:_ → inv:_ →
   Legit (inv .&&. b) c inv →
   Legit inv (While b c) (inv .&&. Not b)
```

Soundness of Floyd-Hoare Logic. We use the above lemmas to establish the soundness of Floyd-Hoare logic as:

```
thmFHLegit :: p:_ → c:_ → q:_ → FH p c q → Legit p c q
```

The implementation of `thm_fh_legit` is an induction (*i.e.,* recursion) over the *structure* of the evidence `FH p c q`, recursively invoking the theorem to inductively assume soundness for the sub-commands, and then using *legitimacy lemmas* that establish legitimacy for that case via the big-step semantics.

5 Algorithmic Verification

The Floyd-Hoare proof rules provide a *methodology* to determine whether a triple is legitimate. However, to do so, we must still construct a valid derivation, which requires us some manual effort to (1) determine where to use the *consequence* rules, and (2) check that various *side conditions* hold for loop invariants.

5.1 Verification Condition Generation

Next, we show how to automate verification by computing a single *verification condition* whose validity demonstrates the legitimacy of a triple.

Weakest Preconditions. In the first step, we *assume* the side conditions hold to check if the given pre-condition establishes the desired post-condition, thereby automating the application of consequence. We will do so via Dijkstra's *predicate transformers* [8]: a function `pre` which given a command `c` and a postcondition `q` computes an assertion `p` corresponding to the *weakest* (*i.e.,* most general) condition under which the `c` is guaranteed to transition to a state at which `q` holds.

```
pre :: Com → Assertion → Assertion
pre Skip          q = q
pre (Assign x a)  q = bsub x a q
pre (Seq c1 c2)   q = pre c1 (pre c2 q)
pre (If b c1 c2)  q = bIte b (pre c1 q) (pre c2 q)
pre (While i _ _) _ = i
```

In the above, `bIte p q r = (p .=> q).&&. (Not p .=> r)` and the substitution `bsub x a q` replaces occurrences of `x` in `q` with `a`.

We can verify a triple $\{p\}\ c\ \{q\}$ by checking the validity of the assertion `p .=>. (pre c q)`. For example to check the triple $\{10 \leq \mathtt{x}\}\ \mathtt{x} \leftarrow \mathtt{x} + 1\ \{10 \leq \mathtt{x}\}$ we would compute the weakest precondition $10 \leq (x + 1)$, and then check the validity of $10 \leq x \Rightarrow 10 \leq (x + 1)$.

Invariant Side Conditions. The definition of `pre` blithely *trusts* the invariants annotations for each `While`-loop are "correct", *i.e.,* are preserved by the loop body and suffice to establish the post-condition. To make the verifier sound, in the second step we must guarantee that the annotations are indeed invariants, by checking them via *invariant side-conditions* computed by the function `ic`

```
ic :: Com → Assertion → Assertion
ic Skip          _ = bTrue
ic (Assign {})   _ = bTrue
ic (Seq c1 c2)   q = ic c1 (pre c2 q) .&&. ic c2 q
ic (If _ c1 c2)  q = ic c1 q .&&. ic c2 q
ic (While i b c) q = ((i .&&. b)     .=>. pre c i) .&&.
                     ((i .&&. Not b) .=>. q      ) .&&.
                     ic c i
```

In essence, `ic` traverses the entire command to find additional constraints that enforce that, at each loop `While i b c` the annotation `i` is indeed an invariant that can be used to find a valid Floyd-Hoare proof for `c`, as made precise by the following lemma

```
lemIC :: c:_ → q:_ → Valid (ic c q) → FH (pre c q) c q
```

That is, we can prove by induction on the structure of `c` that whenever the side-condition holds, executing command `c` from a state `pre c q` establishes the postcondition `q`.

Verification Conditions. We combine the weakest preconditions and invariant side conditions into a single *verification condition* `vc` whose validity checks the correctness of a Floyd-Hoare triple

```
vc :: Assertion → Com → Assertion → Assertion
vc p c q = (p .=>. pre c q) .&&. ic c q
```

We combine `lem_ic` with the rule of consequence `FHPre` to establish that the validity of the `vc` establishes the existence of a Floyd-Hoare proof

```
lemVC :: p:_ → c:_ → q:_ → Valid (vc p c q) → FH p c q
```

We combine the above with the soundness of Floyd-Hoare derivations `thm_fh_legit` to show that verification conditions demonstrate the legitimacy of triples

```
thmVC :: p:_ → c:_ → q:_ → Valid (vc p c q) → Legit p c q
```

5.2 Embedded Verification

Finally we *embed* the `vc` generation within a typed API, turning the type checker into a domain-specific `verify` function

```
verify :: p:_ → c:_ → q:_ → Valid (vc p c q) → ()
verify :: Assertion → Com → Assertion → Valid → ()
verify _ _ _ _ = ()
```

Only the type signature for `verify` is interesting: it says that `verify p c q ok` type-checks *only* if `ok` demonstrates the *validity* of `vc p c q`, which can be done, simply via the term `\_ → ()` as shown in `pf_valid` (in section 4). Lets put our embedded verifier to work, by using it to check some simple IMP programs

Example: Absolute Value. As a first example, lets write a small branching program that assigns `r` to the *absolute value* of `x`.

```
ex_abs = verify p c q (\_ → ())
  where
    p = bTrue

    c = If (N 0 .≤. V "x")
           ("r" ← V "x")
           ("r" ← N 0 .-. V "x")

    q = (N 0 .≤. V "r") .&&. (V "x" .≤. V "r")
```

Example: Swap. Here's a second example that *swaps* the values held inside `x` and `y` via a sequence of arithmetic operations

```
ex_swap = verify p c q (\_ → ())
  where
    p = (V "x" .==. V "a") .&&. (V "y" .==. V "b")

    c = ("x" ← (V "x" .+. V "y")) @@
        ("y" ← (V "x" .-. V "y")) @@
        ("x" ← (V "x" .-. V "y"))

    q = (V "x" .==. V "b") .&&. (V "y" .==. V "a")
```

Example: Sum. Let us conclude with an example mirroring the one we started with in § 2 – a loop that sums up the numbers from 0 to n. Here, we supply the loop invariant that relates the intermediate values of `r` with the loop index `i` to establish the post condition that states the result holds the (closed-form value of the) summation.

```
ex_sum _ = verify p c q (\_ → ())
 where
  p   = N 0 .≤. V "n"

  c   = ("i" ← N 0) @@
        ("r" ← N 0) @@
        While inv (V "i" .!=. V "n") (
          ("r" ← (V "r" .+. V "i")) @@
          ("i" ← (V "i" .+. N 1))
        )

  q   = N 2 .*. V "r" .==. (V "n" .*. (V "n" .-. N 1))

  inv = p .&&.
        ((N 2 .*. V "r") .==. ((V "i" .-. N 1) .*. V "i"))
```

6 Related Work

Refinement Types. Over the past two decades, several groups have designed refinement based verifiers for several languages, starting with the ML family [33, 9, 24, 17, 4, 27], to Racket [16], Scala [12], C [25], JavaScript [6, 31] and Ruby [15]. We refer the interested reader to [14] for more details on how refinement types.

Specifications over User-defined Functions. Refinements are a particular instance of the more general method of *auto-active* (as opposed to interactive) verifiers which, following the early work Floyd [10] and Hoare [13], work by a combination of VC generation and SMT solving [21]. Other prominent SMT-based verifiers like DAFNY and F* support specifications over user defined functions by encoding their semantics with universally-quantified *axioms* that are instantiated via *triggers* [7]. DAFNY and F* use a notion of *fuel* [1] to limit triggering to some fixed depth. This style of unfolding can be shown to be complete for *sufficiently surjective* recursive catamorphisms over algebraic datatypes, *e.g.,* which compute the length of a list or height of a tree [26].

Embedded Verifiers. The notion of embedding verifiers has been explored in several pieces of closely related work. The LMS-VERIFY system [2] uses Scala's lightweight-modular staging feature to compile high-level contracts and code into low-level systems code which can then be automatically verified via an external SMT-based C verifier [3]. The ARMADA system [20] shows how to embed a custom verification framework for concurrent programs with support for *reduction* based refinement proofs [18] within the DAFNY verifier. Finally, the VALE system [11] shows how to build an embedded verifier for a subset of assembly within F* by writing a verified VC generator in F* and then reducing assembly verification to type checking in the host language by using F*'s support for type-level normalization (computation).

Acknowledgements Many thanks to the reviewers for feedback on early drafts of this work. This work was supported by the NSF grants CCF-1514435, CCF-2120642, CCF-1918573, CCF-1911213, and CCF-1917854, and generous gifts from Microsoft Research.

References

1. N. Amin, K. R. M. Leino, and T. Rompf. Computing with an SMT Solver. In *Tests and Proofs*, 2014.
2. Nada Amin and Tiark Rompf. Lms-verify: abstraction without regret for verified systems programming. In Giuseppe Castagna and Andrew D. Gordon, editors, *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages, POPL 2017, Paris, France, January 18-20, 2017*, pages 859–873. ACM, 2017.
3. Patrick Baudin, François Bobot, David Bühler, Loïc Correnson, Florent Kirchner, Nikolai Kosmatov, André Maroneze, Valentin Perrelle, Virgile Prevosto, Julien Signoles, and Nicky Williams. The dogged pursuit of bug-free C programs: the frama-c software analysis platform. *Commun. ACM*, 64(8):56–68, 2021.
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. R. Chugh, D. Herman, and R. Jhala. Dependent types for javascript. In *OOPLSA*, 2012.
7. David Detlefs, Greg Nelson, and James B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.
8. E. W. Dijkstra. Guarded commands, nondeterminacy, and formal derivation of programs. *Communications of the ACM*, 18(8):453–457, 1975.
9. J. Dunfield. Refined typechecking with Stardust. In *PLPV*, 2007.
10. R.W. Floyd. Assigning meanings to programs. In *Mathematical Aspects of Computer Science*. 1967.
11. Aymeric Fromherz, Nick Giannarakis, Chris Hawblitzel, Bryan Parno, Aseem Rastogi, and Nikhil Swamy. A verified, efficient embedding of a verifiable assembly language. *Proc. ACM Program. Lang.*, 3(POPL):63:1–63:30, 2019.
12. Jad Hamza, Nicolas Voirol, and Viktor Kuncak. System FR: formalized foundations for the stainless verifier. *Proc. ACM Program. Lang.*, 3(OOPSLA):166:1–166:30, 2019.
13. C.A.R. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12:576–580, 1969.

14. Ranjit Jhala and Niki Vazou. Refinement types: A tutorial. *Found. Trends Program. Lang.*, 6(3-4):159–317, 2021.
15. Milod Kazerounian, Niki Vazou, Austin Bourgerie, Jeffrey S. Foster, and Emina Torlak. Refinement types for ruby. *CoRR*, abs/1711.09281, 2017.
16. A. M. Kent, D. Kempe, and S. Tobin-Hochstadt. Occurrence typing modulo theories. In *PLDI*, 2016.
17. K. W. Knowles and C. Flanagan. Hybrid type checking. In *TOPLAS*, 2010.
18. Bernhard Kragl, Shaz Qadeer, and Thomas A. Henzinger. Refinement for structured concurrent programs. In Shuvendu K. Lahiri and Chao Wang, editors, *Computer Aided Verification - 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21-24, 2020, Proceedings, Part I*, volume 12224 of *Lecture Notes in Computer Science*, pages 275–298. Springer, 2020.
19. Nico Lehmann, Rose Kunkel, Jordan Brown, Jean Yang, Niki Vazou, Nadia Polikarpova, Deian Stefan, and Ranjit Jhala. STORM: refinement types for secure web applications. In Angela Demke Brown and Jay R. Lorch, editors, *15th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2021, July 14-16, 2021*, pages 441–459. USENIX Association, 2021.
20. Jacob R. Lorch, Yixuan Chen, Manos Kapritsos, Bryan Parno, Shaz Qadeer, Upamanyu Sharma, James R. Wilcox, and Xueyuan Zhao. Armada: low-effort verification of high-performance concurrent programs. In Alastair F. Donaldson and Emina Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 197–210. ACM, 2020.
21. C. G. Nelson. *Techniques for Program Verification*. PhD thesis, Stanford University, 1980.
22. T. Nipkow, L.C. Paulson, and M. Wenzel. *Isabelle/HOL — A Proof Assistant for Higher-Order Logic*. Lecture Notes in Computer Science. 2002.
23. Nadia Polikarpova, Deian Stefan, Jean Yang, Shachar Itzhaky, Travis Hance, and Armando Solar-Lezama. Liquid information flow control. *Proc. ACM Program. Lang.*, 4(ICFP):105:1–105:30, 2020.
24. P. Rondon, M. Kawaguchi, and R. Jhala. Liquid types. In *PLDI*, 2008.
25. P. Rondon, M. Kawaguchi, and R. Jhala. Low-level liquid types. In *POPL*, 2010.
26. P. Suter, A. Sinan Köksal, and V. Kuncak. Satisfiability modulo recursive programs. In *SAS*, 2011.
27. Nikhil Swamy, Cătălin Hriţcu, Chantal Keller, Aseem Rastogi, Antoine Delignat-Lavaud, Simon Forest, Karthikeyan Bhargavan, Cédric Fournet, Pierre-Yves Strub, Markulf Kohlweiss, Jean-Karim Zinzindohoue, and Santiago Zanella-Béguelin. Dependent Types and Multi-Monadic Effects in F*. In *Principles of Programming Languages (POPL)*, 2016.
28. N. Vazou, A. Bakst, and R. Jhala. Bounded refinement types. In *ICFP*, 2015.
29. N. Vazou, E. L. Seidel, R. Jhala, D. Vytiniotis, and S. L. Peyton-Jones. Refinement Types for Haskell. In *ICFP*, 2014.
30. Niki Vazou, Anish Tondwalkar, Vikraman Choudhury, Ryan G. Scott, Ryan R. Newton, Philip Wadler, and Ranjit Jhala. Refinement reflection: complete verification with SMT. *Proc. ACM Program. Lang.*, 2(POPL):53:1–53:31, 2018.
31. P. Vekris, B. Cosman, and R. Jhala. Refinement types for typescript. In *PLDI*, 2016.
32. P. Wadler. Propositions as types. In *Communications of the ACM*, 2015.
33. H. Xi and F. Pfenning. Eliminating array bound checking through dependent types. In *PLDI*, 1998.