Symbolic Types for Lenient Symbolic Execution

STEPHEN CHANG, PLT @ Northeastern University, USA
ALEX KNAUTH, Northeastern University, USA
EMINA TORLAK, University of Washington, USA

We present $\widehat{\lambda}_{\bullet}$, a typed λ-calculus for *lenient symbolic execution*, where some language constructs do not recognize symbolic values. Its type system, however, ensures safe behavior of all symbolic values in a program. Our calculus extends a base occurrence typing system with symbolic types and mutable state, making it a suitable model for both functional and imperative symbolically executed languages. Naively allowing mutation in this mixed setting introduces soundness issues, however, so we further add *concreteness polymorphism*, which restores soundness without rejecting too many valid programs. To show that our calculus is a useful model for a real language, we implemented Typed Rosette, a typed extension of the solver-aided Rosette language. We evaluate Typed Rosette by porting a large code base, demonstrating that our type system accommodates a wide variety of symbolically executed programs.

CCS Concepts: • **Software and its engineering** → *Specialized application languages*;

Additional Key Words and Phrases: Solver-Aided Languages, Symbolic Execution, Type Systems, Macros

ACM Reference Format:
Stephen Chang, Alex Knauth, and Emina Torlak. 2018. Symbolic Types for Lenient Symbolic Execution. *Proc. ACM Program. Lang.* 2, POPL, Article 40 (January 2018), 29 pages. https://doi.org/10.1145/3158128

1 DEBUGGING LENIENT SYMBOLIC EXECUTION

Programmers are increasingly using symbolic execution [Boyer et al. 1975; King 1975], in conjunction with satisfiability solvers, to help find bugs [Cadar et al. 2008; Farzan et al. 2013; Godefroid et al. 2012], verify program properties [Dennis et al. 2006; Jaffar et al. 2012; Near and Jackson 2012], and synthesize code from specifications [Solar-Lezama et al. 2005; Torlak and Bodik 2014]. One challenge when designing such tools is deciding which language constructs should handle symbolic values. Lifting all language constructs is costly to implement and may produce complex solver encodings for some language features. Conversely, restricting symbolic execution to a language subset limits expressive power and the number of possible applications. Some testing frameworks allow a form of mixed symbolic execution where symbolic values are concretized before interacting with unlifted language constructs, but this does not consider all program paths and thus may not be suitable for all applications of symbolic execution.

Ideally, we would like *lenient symbolic execution*, where a small language subset is lifted for symbolic execution but those forms may freely interact with a larger unlifted language [Torlak and Bodik 2013]. This approach considers all paths, is easier to implement, simplifies solver encodings, yet still allows programmers to use expressive language features. Debugging such mixed programs, however, can be tricky. Consider the following pseudocode example:

Authors' addresses: Stephen Chang, CCIS, PLT @ Northeastern University, Boston, MA, USA, stchang@ccs.neu.edu; Alex Knauth, CCIS, Northeastern University, Boston, MA, USA, alexknauth@ccs.neu.edu; Emina Torlak, Paul G. Allen School of Computer Science & Engineering, University of Washington, Seattle, WA, USA, emina@cs.washington.edu.

2475-1421/2018/1-ART40
https://doi.org/10.1145/3158128

```
if (unlifted-int? x) then (add1 x) else (error "cannot add non-integer")
```

The author of this code might reasonably conclude that it will not error when `x` is an integer. Executing the code with `x` as a *symbolic* integer, however, is problematic if `unlifted-int?` does not recognize symbolic values, i.e., it returns true only when given a *concrete* integer and false otherwise. In this scenario, the program will unexpectedly reach the error case. Worse, when symbolic execution is paired with a solver, a programmer may only detect a problem (if at all) after examining the solver's output, and even then will not have much information to debug with.

Thus, despite the benefits, most symbolic execution engines have shunned the mixing of lifted and unlifted language constructs. In contrast, we propose *embracing lenient symbolic execution but supplementing it with a type system*. Such a system would enjoy the benefits of lenient symbolic execution, yet programmers would not be burdened with manually ensuring safe interaction of symbolic values. Further, any safety violations would be automatically detected and reported *before* execution. Finally, such problems are easier to fix since they would be reported in the context of their occurrence, rather than after constraint solving. Our paper makes two main contributions:

(1) $\widehat{\lambda}_{\bullet}$, a typed λ-calculus that ensures safe lenient symbolic execution, and
(2) Typed Rosette, a solver-aided typed programming language based on our calculus.

$\widehat{\lambda}_{\bullet}$ includes symbolic values, higher-order functions, and mutation, and thus may represent the core of both functional and imperative symbolically executed languages. Its type system, in addition to ensuring safe interaction of symbolic values, utilizes occurrence typing and true union types: the path-sensitivity of the former fits well with the path-oriented nature of symbolic execution, and the latter naturally accommodates the symbolic union values that arise when computing with symbolic values. We further extend the system with *concreteness polymorphism*, which utilizes intersection types to add context sensitivity and more precisely tracks the flow of symbolic values, increasing the usefulness of the type system.

To show that our type system is practical to use, we created Typed Rosette, a typed extension of Rosette [Torlak and Bodik 2014], which utilizes symbolic execution to compile programs to solver constraints. Programmers have used Rosette for a variety of verification and synthesis applications [Bornholt and Torlak 2017; Bornholt et al. 2016; Pernsteiner et al. 2016; Phothilimthana et al. 2014; Weitz et al. 2016]. Rosette is an ideal target for our type system since:

- it equips only a subset of its host language, Racket, to handle symbolic values;
- it allows potentially unsafe interaction with the rest of Racket [Flatt and PLT 2010] but discourages programmers from doing so; and
- despite the warnings, users routinely stray from safe core in order to benefit from the extra unsafe features, relying on vigilant programming and detailed knowledge of the language to manually ensure only safe uses of symbolic values.

Typed Rosette aims to help with the last task.

2 ROSETTE PRIMER AND MOTIVATING EXAMPLES

This section introduces untyped Rosette via examples and motivates the need for a type system.

2.1 Restricted (Safe) Rosette

Rosette programmers explicitly define base symbolic values with `define-symbolic*`:

```
#lang rosette/safe ; no lenient symbolic execution
(define-symbolic* x y z integer?)
(and (< x y) (< y z)) ; => symbolic bool (&& (< x y) (< y z))
(if (< x y) 1 (+ z 2)) ; => ⟨[(< x y): 1][¬(< x y): z + 2]⟩
```

Computing with these values may produce other symbolic values, e.g., the third line above evaluates to a symbolic boolean value constructed from the symbolic integers `x`, `y`, and `z`, and the fourth line, due to its symbolic boolean test, explores both branches and merges the results into a guarded symbolic union value.[1] In safe Rosette, invoked with the `#lang rosette/safe` directive on the first line, all language forms are lifted to consume and produce symbolic values when appropriate.

Execution of a program also yields assertions that are passed to a solver, which programmers interact with via Rosette's solver API. For instance, a `solve` query produces an assignment of values to symbolic variables that satisfies the assertions collected during symbolic execution, if one exists:

```
(solve (assert (= x 3))) ; => model: x = 3
```

In this example, the solver determines a value of 3 for the symbolic value `x`. Dually, `verify` tries to find a counterexample to the assertions, e.g.:

```
#lang rosette/safe
(define (sorted? v) ; ascending order
  (define-symbolic* i j integer?)
  (define max (sub1 (vector-length v))) ; max index
  (implies (and (<= 0 i max) (<= 0 j max) (< i j))  ; if valid pair of indices
           (<= (vector-ref v i) (vector-ref v j)))) ; check if pair is sorted
(verify (assert (sorted? (vector 3 5 4)))) ; ✗: i = 1, j = 2
```

This `sorted?` predicate specifies a sortedness condition for vectors. Specifically, a vector is sorted if every pair of elements, represented with symbolic integer indices `i` and `j`, is sorted. In this case, `(vector 3 5 4)` is not sorted and the solver finds a counterexample when $i = 1$ and $j = 2$.

Symbolic values are first-class values that may be passed around or stored in data structures, e.g.:

```
#lang rosette/safe
(define-symbolic* x y z integer?)
(verify #:assume    (assert (and (< x y) (< y z)))
        #:guarantee (assert (sorted? (vector x y z)))) ; ✓ (no counterexamples)
```

This usage of symbolic values enables verifying properties of (concrete) data structures. The vector above contains three symbolic integers `x`, `y`, and `z`. Then, assuming `x < y` and `y < z`, verifying sortedness with the `sorted?` predicate succeeds. Of course, assertions are more typically generated by the program rather than explicitly stated, e.g., here is a basic insertion sort algorithm for lists:

```
#lang rosette/safe
(define (inssort lst) ; insertion sort for lists
  (if (null? lst)
      lst
      (insert (first lst) (inssort (rest lst)))))
(define (insert x lst) ; add x at pos i such that for all j < i, lst[j] < x
  (if (null? lst)
      (list x)
      (if (< x (first lst))
          (cons x lst)
          (cons (first lst) (insert x (rest lst))))))
(define-symbolic* x y z integer?)
(verify (assert (sorted? (lst->vec (inssort (list x y z)))))) ; ✓
```

Querying the solver here confirms that the algorithm satisfies `sorted?` for any three-element list.

[1] Rosette further distinguishes between "solvable" and general symbolic union values but this paper ignores the distinction.

2.2 Full (Unsafe) Rosette

Section 2.1's examples use safe Rosette, where all language forms are lifted to recognize symbolic values. The safe language is limited to a small subset of Racket, however, so programmers frequently use the full Rosette language instead, which includes the remaining features from Racket, unlifted.

Using the full language has two main benefits. The *first* is that programmers enjoy the convenience of a practical, "batteries-included" language. For example, the safe insertion sort code manually deconstructs lists; in the full language, a programmer can utilize pattern matching instead:

```
#lang rosette ; full Rosette includes unlifted features, i.e., allows lenient symbolic execution
(define/match (inssort lst)
  [('()) lst]
  [((cons x xs)) (insert x (inssort xs))])
(define/match (insert x lst)
   [(_ '()) (list x)]
   [(_ (cons y ys)) (if (< x y) (cons x lst) (cons y (insert x ys)))])
(define-symbolic* x y z integer?)
(verify (assert (sorted? (lst->vec (inssort (list x y z)))))) ;✓
```

Even though the matching constructs internally call unlifted list accessors and predicates, their use is acceptable here since symbolic values never reach the unlifted positions. Thus programmers benefit from a more concise implementation of `inssort`.

2.3 Full Rosette Pitfalls

A *second* benefit of using the full language is that its extra features enable more applications of symbolic execution. For example, we can change our predicate to verify "sortedness" of hash tables, which are unavailable in the safe language:

```
#lang rosette
(define (sorted-hash? h)
  (define-symbolic* i j integer?)
  (define max (sub1 (hash-count h)))
  (implies (and (<= 0 i max) (<= 0 j max) (< i j)) ; if valid pair of indices
           (<= (hash-ref h i) (hash-ref h j))))    ; check if pair is sorted
```

The benefits of the full Rosette language, however, come with a tradeoff, as the documentation warns: "Rosette cannot, in general, guarantee the safety of programs that use unlifted constructs. As a result, the programmer is responsible for manually ensuring correctness", where "correctness" means that symbolic values should not reach positions that cannot handle them. Unfortunately, reasoning about such symbolic value flow can be tedious and subtle. Worse, the programmer often has little information with which to debug their incorrect programs, e.g., calling `verify` with the hash predicate returns a nonsensical counterexample:

```
(verify (assert (sorted-hash? (lst->hash (inssort (list x y z)))))) ;✗: i=0, j=1
```

The solver output seems to say that the values at indices `i` and `j` are never sorted. This is especially confusing since replacing `(list x y z)` with a concrete list such as `(list 3 2 1)` produces a sorted list, refuting the solver result. Thus, the programmer is faced with the undesirable choice of tediously tracing the flow of symbolic values by hand in order to check the correctness of their programs, or giving up altogether and going back to the restricted language.

2.4 Introducing Typed Rosette

Typed Rosette reports a type error when symbolic values reach unlifted positions. In section 2.3's example, it turns out that unlifted `hash-ref` is the problem:

```
[ln6,col16]: hash-ref: type mismatch: expected Int, given Int
    expression: i
    in: (hash-ref h i)
```

Once we have located the problem, we can focus on fixing it. In general, a programmer can safely use unlifted functions to traverse and process data structures, so long as any symbolic results do not get used internally within those functions. Section 2.3's example does not satisfy this requirement since it uses `hash-ref` with a symbolic key to essentially "iterate" over the table. Instead, a programmer can use a comprehension form to safely create the desired constraints:

```
#lang typed/rosette
(define (sorted-hash? [h : (HashTable Nat Int)]) -> Bool
  (let ([size (hash-count h)])
    ; hash h is "sorted" if, for all pairs of concrete keys i and j, i < j implies h[i] <= h[j]
    (for*/fold ([result #t]) ([i (in-range size)] [j (in-range size)])
      (and (implies (< i j) (<= (hash-ref h i) (hash-ref h j)))
           result))))
(verify (assert (sorted-hash? (lst->hash (inssort (list x y z)))))) ; ✓
```

Here, `for*/fold` iterates over all pairs `i` and `j`, accumulating constraints in `result`. Even though `for*/fold` is unlifted and unavailable in restricted Rosette, it is used safely here and thus saves the programmer the effort of manually considering all pairs. In addition, we have used the full language to verify a property involving hash tables, which was impossible in the restricted language.[2]

In short, with Typed Rosette programmers may enjoy the benefits of the full Rosette language, yet at the same time avoid the difficult task of tracking and debugging symbolic value flow.

3 A SYMBOLIC TYPED λ-CALCULUS

This section presents $\widehat{\lambda}_{\bullet}$, a typed λ-calculus whose types distinguish symbolic and concrete values. $\widehat{\lambda}_{\bullet}$ builds on Tobin-Hochstadt and Felleisen [2010]'s occurrence typing system (explained in § 3.1), whose path-sensitivity fits well with the path-based nature of symbolic execution. Further, occurrence typing easily accommodates true union types, which are suitable for the union values created during symbolic execution. We first extend this base calculus with symbolic values and types (§ 3.2). Then, to improve the precision, we introduce the notion of *concreteness polymorphism* (§ 3.3). Occurrence typing again comes in handy here, allowing us to dispatch based on the concreteness of a particular value. We further extend the functional core with mutation (§ 3.4), enabling our calculus to model both functional and imperative symbolic execution. Adding mutation naively, however, introduces unsoundness depending on the concreteness of the path so, finally, we extend concreteness polymorphism with additional path-sensitivity and context-sensitivity (§ 3.5) that fixes the possible unsoundness yet preserves enough precision so that mutation remains useful.

3.1 Occurrence Typing, in a Nutshell

Occurrence typing uses the notion of type refinement [Freeman and Pfenning 1991] to add more path-sensitivity than traditional type systems. Specifically, conditionals in an occurrence typing system may refine the types of variables in its branches based on the result of its test:[3]

[2]Section 6.3's example also leverages full Rosette and this same traversal pattern to synthesize incremental algorithms.
[3]For consistency with the rest of the paper, we continue to use Typed Rosette's syntax for most examples.

$$e \in Exp ::= i \mid s \mid \texttt{true} \mid \texttt{false} \mid x \mid op \mid \lambda x{:}\tau\,.\,e \mid e\,e \mid \texttt{if}\,e\,e\,e, \quad i \in \mathbb{Z}, s \in Strings \quad \text{(terms)}$$

$$op \in Op ::= \texttt{bool?} \mid \texttt{not} \mid \texttt{int?} \mid \texttt{add1} \mid \texttt{str?} \mid \texttt{strlen} \quad \text{(primitive ops)}$$

$$\tau \in Ty ::= \texttt{Int} \mid \texttt{String} \mid \texttt{True} \mid \texttt{False} \mid \tau_{fn} \mid \tau \cup \tau \mid \texttt{Any} \quad \text{(types)}$$

$$\tau_{fn} \in TyFn ::= x{:}\tau \xrightarrow{\psi\mid\psi;o} \tau \quad \text{(function types)}$$

$$\psi \in Prop ::= x{:}\tau \mid \neg x{:}\tau \mid \psi \supset \psi \mid \psi \vee \psi \mid \psi \wedge \psi \mid \top \mid \bot \quad \text{(propositions)}$$

$$\Gamma \in Env ::= \psi \ldots, \quad o \in Obj ::= x \mid \cdot \quad \text{(environments and target objects)}$$

$$\text{Abbreviations:} \quad \texttt{Bool} = \texttt{True} \cup \texttt{False}, \quad \texttt{Bot} = \cup$$

Fig. 1. Syntax of base occurrence typing system, based on Tobin-Hochstadt and Felleisen [2010].

```
(define (f [x : (U Int String)]) -> Int
  (if (int? x)
      (add1 x)
      (strlen x)))
```

The x input in the above function definition may initially be either a string or integer. Its type, however, is refined to Int in the "then" branch and String in the "else" branch, and thus both the addition and string length computations type check and the function returns an Int.

The conditional test expression dictates the type refinements in the branches. More precisely, type checking judgements have shape $\Gamma \vdash e \ : \ \tau;\ \psi^+ \mid \psi^-$ where, if e is used as a conditional test, then ψ^+ and ψ^- are logical propositions that describe how to refine types in the "then" and "else" branches, respectively. In the example, type checking `(int? x)` computes $\psi^+ = x{:}\texttt{Int}$ and $\psi^- = \neg x{:}\texttt{Int}$. When these constraints are combined with with x's original type `(U Int String)`, the type checker may conclude $x{:}\texttt{Int}$ in the "then" branch and $x{:}\texttt{String}$ in the "else" branch.

Formally, figure 1 presents the syntax for a basic occurrence typing system; it more or less resembles the calculus of Tobin-Hochstadt and Felleisen [2010]. The language includes three types of literal values: integers, strings, and booleans. Integers and strings have base types Int and String, respectively. To keep our type judgements uniform, if conditionals use "non-false" semantics, i.e., its test expression may have any type and all non-false values are considered equivalent to true.[4] Consequently the type system includes two separate Boolean base types, True and False; we use an abbreviation Bool for the union of True and False when needed. The rest of the expressions are variables, primitive operations, lambdas, and function application; the rest of the types are function types, unions, and Any. Unions are a multi-arity type constructor, e.g., a union of no arguments is equivalent to the empty type, but we present it syntactically as a binary constructor when it's more convenient. Finally, Any includes all the other types.

$\widehat{\lambda}_o$'s function type differs from the standard arrow in three ways: it (1) binds a parameter that is in scope for the rest of the type, (2) specifies two refinement propositions ψ, and (3) specifies a "target object" o. The section's earlier example introduced how type checking a conditional's test additionally computes propositions which then refine a variable's type in the branches. The origin of these refinements are predicate functions such as int?, which has type $x{:}\texttt{Any} \xrightarrow{x:\texttt{Int}\mid\neg x:\texttt{Int}} \texttt{Bool}$. In other words applying int? reveals additional information about its argument: it either is an integer or is not an integer. A base type environment lookup function δ_τ, defined in figure 2 for a few examples, assigns types to other primitive operations in a similar manner. The type of the add1 and strlen functions specify $\top$ and $\bot$ propositions because their inputs, integers and strings,

[4]The alternative requires separating boolean expression judgements, which include refinements, and non-boolean ones.

$$\delta_\tau\ \texttt{int?} = x{:}\texttt{Any} \xrightarrow{x:\texttt{Int}\mid\neg x:\texttt{Int}} \texttt{Bool} \qquad \boxed{\delta_\tau : Op \to TyFn}$$

$$\delta_\tau\ \texttt{str?} = x{:}\texttt{Any} \xrightarrow{x:\texttt{String}\mid\neg x:\texttt{String}} \texttt{Bool}$$

$$\delta_\tau\ \texttt{not} = x{:}\texttt{Any} \xrightarrow{x:\texttt{False}\mid\neg x:\texttt{False}} \texttt{Bool}$$

$$\delta_\tau\ \texttt{add1} = x{:}\texttt{Int} \xrightarrow{\top\mid\bot} \texttt{Int}$$

$$\delta_\tau\ \texttt{strlen} = x{:}\texttt{String} \xrightarrow{\top\mid\bot} \texttt{Int}$$

Fig. 2. Types for primitive ops in base occurrence typing system.

respectively, are always considered "true" when used in a conditional test, and the result of these primitive functions do not reveal additional information about the types of their inputs.

The final component of the function type is a target object o. Target objects, specified as the final component of a typing judgement $\Gamma \vdash e : \tau;\ \psi \mid \psi;\ o$, add flow-sensitivity to the type system in order to track the variable that "would be" modified by the ψ refinements. In the previous example, the result of `(int? x)` straightforwardly determines whether `x` is an integer. The target of refinement may be less straightforward, however, if `int?` is applied to a non-variable expression. For example if the conditional were changed to `(if (int? (id x)) .... )` where `id` is the identify function, the type system should still know that `x` is the target of refinement, even though it passes through a function call.[5] To address these cases, function types also specify a target object. For example `id` would have type $x : \texttt{Any} \xrightarrow{\neg x:\texttt{False}\mid x:\texttt{False};\ x} \texttt{Any}$, where the object component x specifies that the function input becomes the target object after the function is applied. Together, the propositions and target objects make occurrence typing fully compositional. In other words, a conditional will properly refine the types in its branches with any arbitrary expression as its test.

Figure 3 presents the type rules for figure 1's base occurrence typing language.[6] When integers, strings, and other non-false values are used as a conditional test, the proposition environment is unchanged in "then" branch, denoted with proposition $\top$, and may be used to prove any conclusion in the "else" branch, denoted with $\bot$. Dually, the rule for false values flips these propositions.

Thus far we have only informally described propositions and how conditionals use them to refine types in their branches. Formally, a proposition environment Γ, which generalizes the type environment found in conventional type systems, propagates these propositions. The propositions are used to define the proof system in figure 4, and the T-Var rule in figure 3 uses this proof system to determine the type of a variable. The T-Var rule also states that if a variable is used as a conditional test, the only thing we can conclude in the "then" branch is that the variable is not false. Dually, the variable must be false in the "else" branch. Finally, a variable reference sets the current refinement target object to be that variable.

The proof system used to determine a variable's type mostly consists of the standard rules of propositional logic; figure 4 shows a few of the rules. The additional L-Restrict and L-Remove rules combine various propositions to refine a more general type into a more precise one. The *restrict* and *remove* metafunctions, defined in figure 6, roughly correspond to set intersection and set difference operations, respectively. Revisiting the example from the beginning of the section, if the proposition

[5]The "target object" component is particularly important when dealing with data structures, as originally presented by Tobin-Hochstadt and Felleisen [2010]. We omit data structures to simplify presentation of our type system, since they are orthogonal to our goal of symbolic types, but they should work in the same manner described in the original calculus.
[6]We may omit showing the object component in function types, e.g., in figure 2, and type rules when they have the $\cdot$ object.

$$\textsc{T-Int}\ \ \Gamma \vdash i \ : \ \texttt{Int};\ \top \mid \bot \qquad \textsc{T-True}\ \ \Gamma \vdash \texttt{true} \ : \ \texttt{True};\ \top \mid \bot \qquad \textsc{T-False}\ \ \Gamma \vdash \texttt{false} \ : \ \texttt{False};\ \bot \mid \top$$

$$\textsc{T-String}\ \ \Gamma \vdash s \ : \ \texttt{String};\ \top \mid \bot \qquad \textsc{T-Prim}\ \ \Gamma \vdash op \ : \ \delta_\tau(op);\ \top \mid \bot$$

$$\textsc{T-Var}\ \ \frac{\Gamma \vdash_\psi x : \tau}{\Gamma \vdash x \ : \ \tau;\ \neg x{:}\texttt{False} \mid x{:}\texttt{False};\ x}$$

$$\textsc{T-If}\ \ \frac{\begin{array}{c}\Gamma \vdash e_1 \ : \ \tau_1;\ \psi_1^+ \mid \psi_1^- \\ \Gamma, \psi_1^+ \vdash e_2 \ : \ \tau;\ \psi_2^+ \mid \psi_2^-;\ o \\ \Gamma, \psi_1^- \vdash e_3 \ : \ \tau;\ \psi_3^+ \mid \psi_3^-;\ o\end{array}}{\Gamma \vdash \texttt{if}\, e_1\, e_2\, e_3 \ : \ \tau;\ \psi_2^+ \vee \psi_3^+ \mid \psi_2^- \vee \psi_3^-;\ o}$$

$$\textsc{T-Lam}\ \ \frac{\Gamma, x{:}\tau \vdash e \ : \ \tau';\ \psi^+ \mid \psi^-;\ o}{\Gamma \vdash \lambda x{:}\tau\,.\,e \ : \ x{:}\tau \xrightarrow{\psi^+ \mid \psi^-;\, o} \tau';\ \top \mid \bot}$$

$$\textsc{T-App}\ \ \frac{\Gamma \vdash e_1 \ : \ x{:}\tau_1 \xrightarrow{\psi^+ \mid \psi^-;\, o} \tau_2;\ \psi_1^+ \mid \psi_1^- \qquad \Gamma \vdash e_2 \ : \ \tau_1;\ \psi_2^+ \mid \psi_2^-;\ o_2}{\Gamma \vdash e_1\, e_2 \ : \ \tau_2[x := o_2];\ \psi^+[x := o_2] \mid \psi^-[x := o_2];\ o[x := o_2]}$$

$$\textsc{T-Subsume}\ \ \frac{\Gamma \vdash e \ : \ \tau;\ \psi^+ \mid \psi^-;\ o \qquad \Gamma, \psi^+ \vdash_\psi \psi^{+\prime} \qquad \Gamma, \psi^- \vdash_\psi \psi^{-\prime} \qquad \tau <: \tau' \qquad o <: o'}{\Gamma \vdash e \ : \ \tau';\ \psi^{+\prime} \mid \psi^{-\prime};\ o'}$$

Fig. 3. Type rules for base occurrence typing system.

$$\textsc{L-Atom}\ \ \frac{\psi \in \Gamma}{\Gamma \vdash_\psi \psi} \qquad \textsc{L-True}\ \ \Gamma \vdash_\psi \top \qquad \textsc{L-False}\ \ \frac{\Gamma \vdash_\psi \bot}{\Gamma \vdash_\psi \psi} \qquad \textsc{L-Sub}\ \ \frac{\Gamma \vdash_\psi x{:}\tau_1 \qquad \tau_1 <: \tau_2}{\Gamma \vdash_\psi x{:}\tau_2} \qquad \textsc{L-NotSub}\ \ \frac{\Gamma \vdash_\psi \neg x{:}\tau_2 \qquad \tau_1 <: \tau_2}{\Gamma \vdash_\psi \neg x{:}\tau_1}$$

$$\textsc{L-OrI}\ \ \frac{\Gamma \vdash_\psi \psi_1 \text{ or } \Gamma \vdash_\psi \psi_2}{\Gamma \vdash_\psi \psi_1 \vee \psi_2} \qquad \textsc{L-OrE}\ \ \frac{\Gamma, \psi_1 \vdash_\psi \psi \qquad \Gamma, \psi_2 \vdash_\psi \psi}{\Gamma, \psi_1 \vee \psi_2 \vdash_\psi \psi} \qquad \textsc{L-Restrict}\ \ \frac{\Gamma \vdash_\psi x{:}\tau_1 \qquad \Gamma \vdash_\psi x{:}\tau_2}{\Gamma \vdash_\psi x{:}(\mathit{restrict}\,\tau_1\,\tau_2)} \qquad \textsc{L-Remove}\ \ \frac{\Gamma \vdash_\psi x{:}\tau_1 \qquad \Gamma \vdash_\psi \neg x{:}\tau_2}{\Gamma \vdash_\psi x{:}(\mathit{remove}\,\tau_1\,\tau_2)}$$

Fig. 4. A few proof rules for computing a variable's type in base occurrence typing system.

$$\textsc{Sub-}\cup\textsc{-L}\ \ \frac{\tau_1 <: \tau \qquad \tau_2 <: \tau}{\tau_1 \cup \tau_2 <: \tau} \qquad \textsc{Sub-}\cup\textsc{-R}\ \ \frac{\tau <: \tau_1 \text{ or } \tau <: \tau_2}{\tau <: \tau_1 \cup \tau_2} \qquad \textsc{Sub-arr}\ \ \frac{\tau_3 <: \tau_1 \qquad \tau_2 <: \tau_4 \qquad \psi_1^+ \vdash \psi_2^+ \qquad \psi_1^- \vdash \psi_2^-}{x{:}\tau_1 \xrightarrow{\psi_1^+ \mid \psi_1^-;\, o_1} \tau_2 \ <: \ x{:}\tau_3 \xrightarrow{\psi_2^+ \mid \psi_2^-;\, o_2} \tau_4}$$

Fig. 5. A few subtyping rules for base occurrence typing system.

environment contains $x{:}\texttt{Int} \cup \texttt{String}$ and $x{:}\texttt{Int}$, like in the "then" branch", then the L-Restrict rule allows the conclusion $x{:}\texttt{Int}$. If the proposition environment contains $x{:}\texttt{Int} \cup \texttt{String}$ and $\neg x{:}\texttt{Int}$, like in the "else" branch", then the L-Remove rule allows the conclusion $x{:}\texttt{String}$.

Returning to figure 3, the T-If rule shows how the positive and negative propositions from the test expression are propagated to the branches of a conditional, as previously described. The

$$\boxed{restrict : Ty\ Ty \to Ty}$$

$$\begin{aligned} restrict\,\tau_1\,\tau_2 &= \texttt{Bot}, \text{ if } \tau_1 \not<: \tau_2, \tau_2 \not<: \tau_1 \\ restrict\,(\tau_1 \cup \tau_2)\,\tau &= (restrict\,\tau_1\,\tau) \cup (restrict\,\tau_2\,\tau) \\ restrict\,\tau_1\,\tau_2 &= \tau_1, \text{ if } \tau_1 <: \tau_2 \\ restrict\,\tau_1\,\tau_2 &= \tau_2, \text{ otherwise} \end{aligned}$$

$$\boxed{remove : Ty\ Ty \to Ty}$$

$$\begin{aligned} remove\,\tau_1\,\tau_2 &= \texttt{Bot}, \text{ if } \tau_1 <: \tau_2 \\ remove\,(\tau_1 \cup \tau_2)\,\tau &= (remove\,\tau_1\,\tau) \cup (remove\,\tau_2\,\tau) \\ remove\,\tau_1\,\tau_2 &= \tau_1, \text{ otherwise} \end{aligned}$$

Fig. 6. Metafunctions for base occurrence typing system.

$$\begin{aligned} e \in Exp &::= \ldots \mid \widehat{x^{\tau}} && \text{(terms)} \\ \tau \in Ty &::= \ldots \mid \widehat{\tau} && \text{(types)} \end{aligned}$$

$$\boxed{concrete? : Ty \to \texttt{Bool}}$$

$$\begin{aligned} concrete?\,\widehat{\tau} &= \texttt{false} \\ concrete?\,\tau_1 \cup \tau_2 &= concrete?\,\tau_1 \text{ and } concrete?\,\tau_2 \\ concrete?\,\tau &= \texttt{true}, \text{ otherwise} \end{aligned}$$

Fig. 7. (left) $\widehat{\lambda_{\bullet}}$, lambda-calculus with symbolic values, extends core occurrence typing calculus from figure 1; (right) Metafunction to determine whether a type represents a concrete value.

propositions for the conditional expression itself then, are disjunctions of the propositions from each branch. The T-Lam rule shows how the latent propositions and target object from a lambda's body are transferred to its function type. Dually, T-App specifies that applying a function uses the propositions and object from the function type as the propositions and object of the application expression, except that the object from the argument replaces the function parameter.

Finally, our base occurrence typing system uses a subtyping relation—a few rules are shown in figure 5— which is used in figure 3's T-Subsume rule. Notably, the Sub-∪-R shows how unions may be introduced, to complement the union-elimination nature of the `if` form. The remaining unshown rules are more or less standard.

3.2 Adding Symbolic Values

We introduce $\widehat{\lambda_{\bullet}}$ by extending the occurrence typing calculus in figure 1 with symbolic values and symbolic types. Specifically, the syntax for $\widehat{\lambda_{\bullet}}$ in figure 7 (left) extends figure 1 with symbolic literal values $\widehat{x^{\tau}}$. Programmers specify the kind of symbolic value they wish to introduce into the program with a type annotation. At the type level, $\widehat{\lambda_{\bullet}}$ distinguishes symbolic values from concrete ones with a $\widehat{\cdot}$ constructor and thus a $\widehat{x^{\tau}}$ value has type $\widehat{\tau}$. $\widehat{\lambda_{\bullet}}$ allows only integer and boolean base symbolic values, as enforced by the T-SymInt and T-SymBool rules in figure 8. Consequently, strings are always concrete and string functions in $\widehat{\lambda_{\bullet}}$ represent the "unlifted constructs" in a language supporting lenient symbolic execution.

The `if` form may also create symbolic values. Consider the expression $\texttt{if}\ \widehat{x^{\texttt{Bool}}}\ 1\ 2$. Since $\widehat{x^{\texttt{Bool}}}$ represents both true and false, symbolic execution must explore both branches and thus the expression evaluates to a symbolic value that is either 1 or 2. Figure 8's T-If-Sym rule accounts for this creation of symbolic values. Specifically, when the test expression is symbolic and *possibly* `False`, then the `if` expression has a symbolic type using $\widehat{\cdot}$. Observe that the "$\texttt{False} <: \tau_1$" premise expresses the "possibly false", which includes expressions that are not completely boolean, like $\texttt{if}\ \widehat{x^{\texttt{Bool}}}\ \widehat{x^{\texttt{Bool}}}\ \widehat{x^{\texttt{Int}}}$. Such a conditional test should still create a symbolic value.

Dually, if a conditional test expression is a non-`false` symbolic value or a concrete value, symbolic execution need only explore one branch and thus T-If-Conc does not apply the $\widehat{\cdot}$ constructor to

$$\textsc{T-SymInt}\ \ \Gamma \vdash \widehat{x^{\texttt{Int}}} \ : \ \widehat{\texttt{Int}};\ \top \mid \bot \qquad \textsc{T-SymBool}\ \ \Gamma \vdash \widehat{x^{\texttt{Bool}}} \ : \ \widehat{\texttt{Bool}};\ \top \mid \top$$

$$\textsc{T-If-Sym}\ \ \frac{\begin{array}{c}\Gamma \vdash e_1 \ : \ \tau_1;\ \psi_1^+ \mid \psi_1^- \\ \textit{symbolic?}\,\tau_1 \text{ and } \texttt{False} <: \tau_1 \\ \Gamma, \psi_1^+ \vdash e_2 \ : \ \tau;\ \psi_2^+ \mid \psi_2^-;\ o \\ \Gamma, \psi_1^- \vdash e_3 \ : \ \tau;\ \psi_3^+ \mid \psi_3^-;\ o\end{array}}{\Gamma \vdash \texttt{if}\, e_1\, e_2\, e_3 \ : \ \widehat{\tau};\ \psi_2^+ \vee \psi_3^+ \mid \psi_2^- \vee \psi_3^-;\ o}$$

$$\textsc{T-If-Conc}\ \ \frac{\begin{array}{c}\Gamma \vdash e_1 \ : \ \tau_1;\ \psi_1^+ \mid \psi_1^- \\ \textit{concrete?}\,\tau_1 \text{ or } (\textit{symbolic?}\,\tau_1 \text{ and } \texttt{False} \not<: \tau_1) \\ \Gamma, \psi_1^+ \vdash e_2 \ : \ \tau;\ \psi_2^+ \mid \psi_2^-;\ o \\ \Gamma, \psi_1^- \vdash e_3 \ : \ \tau;\ \psi_3^+ \mid \psi_3^-;\ o\end{array}}{\Gamma \vdash \texttt{if}\, e_1\, e_2\, e_3 \ : \ \tau;\ \psi_2^+ \vee \psi_3^+ \mid \psi_2^- \vee \psi_3^-;\ o}$$

$$\textsc{Sub-ConcAny}\ \ \frac{\textit{concrete?}\,\tau}{\tau <: \texttt{Any}} \qquad \textsc{Sub-SymAny}\ \ \tau <: \widehat{\texttt{Any}} \qquad \textsc{Sub-Sym}\ \ \tau <: \widehat{\tau}$$

Fig. 8. Some type and subtyping rules for $\widehat{\lambda_{\bullet}}$, a λ-calculus with symbolic values

$$\tau \in Ty ::= \ldots \mid \tau_{fn} \cap \tau_{fn} \qquad op \in Op ::= \ldots \mid \texttt{conc?} \qquad \text{(types and prim ops)}$$

$$\textsc{T-Lam-ConcArg}\ \ \frac{\Gamma, x{:}\tau \vdash e \ : \ \tau';\ \psi^+ \mid \psi^-;\ o}{\Gamma \vdash \lambda x{:}\tau\,.\,e \ : \ x{:}\tau \xrightarrow{\psi^+ \mid \psi^-;o} \tau';\ \top \mid \bot} \qquad \textsc{T-Lam-SymArg}\ \ \frac{\Gamma, x{:}\widehat{\tau} \vdash e \ : \ \tau';\ \psi^+ \mid \psi^-;\ o}{\Gamma \vdash \lambda x{:}\tau\,.\,e \ : \ x{:}\widehat{\tau} \xrightarrow{\psi^+ \mid \psi^-;o} \tau';\ \top \mid \bot}$$

$$\textsc{T-Inter-I}\ \ \frac{\Gamma \vdash e \ : \ \tau_{fn_1} \qquad \Gamma \vdash e \ : \ \tau_{fn_2}}{\Gamma \vdash e \ : \ \tau_{fn_1} \cap \tau_{fn_2}} \qquad \textsc{Sub-}\cap\textsc{-L}\ \ \frac{\tau_{fn_1} <: \tau_{fn} \text{ or } \tau_{fn_2} <: \tau_{fn}}{\tau_{fn_1} \cap \tau_{fn_2} <: \tau_{fn}} \qquad \textsc{Sub-}\cap\textsc{-R}\ \ \frac{\tau_{fn} <: \tau_{fn_1} \qquad \tau_{fn} <: \tau_{fn_2}}{\tau_{fn} <: \tau_{fn_1} \cap \tau_{fn_2}}$$

Fig. 9. $\widehat{\lambda_{\bullet}}$ with concreteness polymorphism, part 1.

the type of the `if` expression. T-If-Sym and T-If-Conc, which replace T-If from figure 3, use a *concrete?* metafunction on types, defined in figure 7 (right), which returns true if its argument type represents concrete values. We also use the abbreviation *symbolic?*τ to mean "not *concrete?*τ".

The $\widehat{\cdot}$ type actually denotes *possibly* symbolic values. That is, concrete and symbolic types are not disjoint but rather form a hierarchy. Thus, a concrete value may have a $\widehat{\cdot}$ type, which may occur when symbolic execution utilizes dynamic information unavailable during type checking. For example, if both branches of a symbolic conditional evaluate to the same concrete value, the result of evaluation should be that concrete value. The type system, however, must conservatively label the expression as possibly symbolic. The Sub-Sym subtype rule in figure 8 specifies this relation. Specifically a type τ is considered a subtype of a possibly symbolic version of that type $\widehat{\tau}$. In addition, $\widehat{\lambda_{\bullet}}$ requires two separate subtype rules to handle the `Any` type: the `Any` type represents only concrete values while $\widehat{\texttt{Any}}$ includes symbolic values.

3.3 Concreteness Polymorphism of Arguments

Adding symbolic values and unlifted constructs that do not handle symbolic values enables $\widehat{\lambda_{\bullet}}$ to model lenient symbolic execution. Adding symbolic *types* allows the type system to ensure the safe behavior of symbolic values by preventing them from reaching unlifted constructs, which should

$$\boxed{\delta_\tau : Op \to Ty}$$

$$\delta_\tau \texttt{int?} = x{:}\texttt{Any} \xrightarrow{x:\texttt{Int}\mid\neg x:\texttt{Int}} \texttt{Bool} \cap x{:}\widehat{\texttt{Any}} \xrightarrow{x:\widehat{\texttt{Int}}\mid\neg x:\widehat{\texttt{Int}}} \widehat{\texttt{Bool}}$$

$$\delta_\tau \texttt{str?} = x{:}\texttt{Any} \xrightarrow{x:\texttt{String}\mid\neg x:\texttt{String}} \texttt{Bool}$$

$$\delta_\tau \texttt{not} = x{:}\texttt{Any} \xrightarrow{x:\texttt{False}\mid\neg x:\texttt{False}} \texttt{Bool} \cap x{:}\widehat{\texttt{Any}} \xrightarrow{x:\widehat{\texttt{False}}\mid\neg x:\widehat{\texttt{False}}} \widehat{\texttt{Bool}}$$

$$\delta_\tau \texttt{add1} = x{:}\texttt{Int} \xrightarrow{\top\mid\bot} \texttt{Int} \cap x{:}\widehat{\texttt{Int}} \xrightarrow{\top\mid\bot} \widehat{\texttt{Int}}$$

$$\delta_\tau \texttt{strlen} = x{:}\texttt{String} \xrightarrow{\top\mid\bot} \texttt{Int}$$

$$\delta_\tau \texttt{conc?} = x{:}\texttt{Any} \xrightarrow{\top\mid\bot} \texttt{True} \cap x{:}\widehat{\texttt{Any}} \xrightarrow{x:\texttt{Any}\mid\neg x:\texttt{Any}} \widehat{\texttt{Bool}}$$

Fig. 10. Types for primitive ops in $\widehat{\lambda}_{\bullet}$.

be assigned concrete types. To fully benefit from lenient symbolic execution, however, unlifted constructs should be allowed to interact with *concrete* values as much as possible. To achieve this, the type system must preserve concreteness at the type level as much as possible.

Naively type checking symbolic values, however, quickly causes the entire program to become symbolic. This limits the usefulness of lenient symbolic execution because it rejects too many safe programs. As an example, what type should $\widehat{\lambda}_{\bullet}$ assign to `add1`? It should accommodate symbolic values and thus one might naively assign type $\widehat{\texttt{Int}} \to \widehat{\texttt{Int}}$. With this type, however, the result of applying `add1` to a *concrete* value would have a symbolic type and thus could not be used with an unlifted arithmetic function even though it is perfectly safe. To improve the precision of $\widehat{\lambda}_{\bullet}$'s type system, i.e., to preserve concreteness as much as possible, we use intersection function types, shown in figure 9. We call this kind of finitary polymorphism *concreteness polymorphism*.

Concreteness polymorphism enables more precise types for primitive operations as shown in figure 10. In contrast with figure 2, figure 10 assigns types that allows boolean and integer primitive functions to handle symbolic values. These lifted operations with intersection types may be applied at any of their constituent function types, as specified by the Sub-∩-L rule in figure 9 (in conjunction with T-Subsume and T-App). As a result the type checker safely allows the result of `add1` to be passed to an unlifted arithmetic operation if the original argument was concrete. Primitive string functions remain unlifted in $\widehat{\lambda}_{\bullet}$ and thus do not accept symbolic values. Thus applying either `str?` or `strlen` to a symbolic value results in a type error. Finally, $\widehat{\lambda}_{\bullet}$ adds a `conc?` primitive which, in conjunction with `if` and occurrence typing, serves as the elimination rule for values with possibly-symbolic types. In other words, it allows programmers to write their own functions whose behavior may depend on the concreteness of its inputs.

Lambdas support concreteness polymorphism as well. Specifically, lambda bodies are type checked twice: once with a concrete input and once with a symbolic input, as specified by T-Lam-ConcArg and T-Lam-SymArg, respectively, in figure 9 (these rules replace T-Lam in figure 3).[7] The T-Inter-I rule allows assigning a lambda an intersection type consisting of these two types. (Note that a separate T-Inter-I rule is necessary since Sub-∩-R is insufficient for introducing the intersection type in this case.)

[7] Of course, to avoid exponential explosion of function type sizes, a practical language would allow programmers to more precisely choose which combinations of parameter concreteness should be included in the type.

$$e \in Exp ::= \ldots \mid () \mid \texttt{set!}\, x\, e \mid e\,;e \qquad \tau \in Ty ::= \ldots \mid \texttt{Unit} \qquad \text{(terms and types)}$$

Fig. 11. $\widehat{\lambda_{\bullet}}$ symbolic lambda-calculus, with mutation

3.4 Adding Mutation

To allow $\widehat{\lambda_{\bullet}}$ to serve as a model for imperative symbolically executed languages as well, we next add mutation, specifically `set!` and sequencing expressions, and a `Unit` type, as shown in figure 11. A naive combination of symbolic paths and mutation, however, is unsound. For example, a standard type rule for `set!` might look like:

$$\textsc{Mut-Wrong}\ \frac{x{:}\tau \in \Gamma \qquad \Gamma \vdash e\ :\ \tau}{\Gamma \vdash \texttt{set!}\, x\, e\ :\ \texttt{Unit}}$$

This may not be safe, as seen in the following example:

```
(define-symbolic* b boolean?)
(define x 0)          ; x is a concrete value with concrete type Int
(if b (set! x 10) (set! x 11)
x ; => ⟨[b: 10][¬b: 11]⟩  ; x is a symbolic value, but still has concrete type Int
```

The example mutates x under a symbolic path, changing it from a concrete to a symbolic value. The type remains concrete, however, and thus the operation is unsound. Observe that x becomes bound to a symbolic value, even though it is only ever assigned concrete values. One way to restore safety is to force all mutable variables to have symbolic types but this conflicts with our goal of preserving concreteness as much as possible for lenient symbolic execution.

Instead, our type system tracks the concreteness of the path and `set!` rule looks something like:

$$\textsc{Mut-OK}\ \frac{x{:}\tau \in \Gamma \qquad \Gamma \vdash e\ :\ \tau \qquad \textit{symbolic?}\,\tau \text{ OR path is concrete}}{\Gamma \vdash \texttt{set!}\, x\, e\ :\ \texttt{Unit}}$$

The rule allows mutation if the value has symbolic type; if the value is concrete, mutation is allowed only if the path is also concrete. This extra path sensitivity during type checking, however, requires additional context sensitivity in function calls. For example, the following `set!` in the λ body may be safe or unsafe depending on its call site:

```
(define x 0) ; x is a concrete value with concrete type Int
(define (f [y : Int]) (set! x y))
(f 1)
x ; SAFE: x still concrete with concrete type
(define-symbolic* b : Bool)
(if b (f 2) (f 3)))
x ; => ⟨[b: 2][¬b: 3]⟩ (UNSAFE: x is a symbolic val but has concrete type)
```

The type system should ideally allow defining `f` since it may be used safely, but disallow calls to `f` in unsafe contexts. In other words, the type system should reject the expressions `(f 2)` and `(f 3)` above. To achieve this, we extend our notion of concreteness polymorphism to include the concreteness of the path.

$$\tau_{fn} ::= \pi; x{:}\tau \xrightarrow{\psi\,|\,\psi} \tau \quad \text{(function types)}$$
$$\pi \in Path ::= \bullet \mid \circ \quad \text{(path condition)}$$

Sub-Path
$\bullet <: \circ$

Fig. 12. (left) $\widehat{\lambda_{\bullet}}$ concreteness polymorphic function types; (right) subtype relation for paths

$$\textsc{T-Lam-ConcArgPath}^{\bullet}\ \frac{\Gamma, x{:}\tau \overset{\bullet}{\vdash} e : \tau';\ \psi^+ \mid \psi^-}{\Gamma \overset{\bullet}{\vdash} \lambda x{:}\tau.e : \bullet; x{:}\tau \xrightarrow{\psi^+|\psi^-} \tau',}$$

$$\textsc{T-Lam-ConcArg-SymPath}^{\bullet}\ \frac{\Gamma, x{:}\tau \overset{\circ}{\vdash} e : \tau';\ \psi^+ \mid \psi^-}{\Gamma \overset{\bullet}{\vdash} \lambda x{:}\tau.e : \circ; x{:}\tau \xrightarrow{\psi^+|\psi^-} \tau',}$$

$$\textsc{T-Lam-SymArg-ConcPath}^{\bullet}\ \frac{\Gamma, x{:}\widehat{\tau} \overset{\bullet}{\vdash} e : \tau';\ \psi^+ \mid \psi^-}{\Gamma \overset{\bullet}{\vdash} \lambda x{:}\tau.e : \bullet; x{:}\widehat{\tau} \xrightarrow{\psi^+|\psi^-} \tau',}$$

$$\textsc{T-Lam-SymArgPath}^{\bullet}\ \frac{\Gamma, x{:}\widehat{\tau} \overset{\circ}{\vdash} e : \tau';\ \psi^+ \mid \psi^-}{\Gamma \overset{\bullet}{\vdash} \lambda x{:}\tau.e : \circ; x{:}\widehat{\tau} \xrightarrow{\psi^+|\psi^-} \tau',}$$

$$\textsc{T-Lam-ConcArgPath}^{\circ}\ \frac{\Gamma, x{:}\tau \overset{\bullet}{\vdash} e : \tau';\ \psi^+ \mid \psi^-}{\Gamma \overset{\circ}{\vdash} \lambda x{:}\tau.e : \bullet; x{:}\tau \xrightarrow{\psi^+|\psi^-} \tau',}$$

$$\textsc{T-Lam-ConcArg-SymPath}^{\circ}\ \frac{\Gamma, x{:}\tau \overset{\circ}{\vdash} e : \tau';\ \psi^+ \mid \psi^-}{\Gamma \overset{\circ}{\vdash} \lambda x{:}\tau.e : \circ; x{:}\tau \xrightarrow{\psi^+|\psi^-} \tau',}$$

$$\textsc{T-Lam-SymArg-ConcPath}^{\circ}\ \frac{\Gamma, x{:}\widehat{\tau} \overset{\bullet}{\vdash} e : \tau';\ \psi^+ \mid \psi^-}{\Gamma \overset{\circ}{\vdash} \lambda x{:}\tau.e : \bullet; x{:}\widehat{\tau} \xrightarrow{\psi^+|\psi^-} \tau',}$$

$$\textsc{T-Lam-SymArgPath}^{\circ}\ \frac{\Gamma, x{:}\widehat{\tau} \overset{\circ}{\vdash} e : \tau';\ \psi^+ \mid \psi^-}{\Gamma \overset{\circ}{\vdash} \lambda x{:}\tau.e : \circ; x{:}\widehat{\tau} \xrightarrow{\psi^+|\psi^-} \tau',}$$

$$\textsc{T-App}^{\pi}\ \frac{\Gamma \overset{\pi}{\vdash} e_1 : \pi; x{:}\tau_1 \xrightarrow{\psi^+|\psi^-} \tau_2 \qquad \Gamma \overset{\pi}{\vdash} e_2 : \tau_1}{\Gamma \overset{\pi}{\vdash} e_1\, e_2 : \tau_2;\ \psi^+ \mid \psi^-}$$

$$\textsc{T-If-Conc}^{\pi}\ \frac{\begin{array}{c}\Gamma \overset{\pi}{\vdash} e_1 : \tau_1;\ \psi_1^+ \mid \psi_1^- \\ \textit{concrete?}\,\tau_1 \text{ or } (\textit{symbolic?}\,\tau_1 \text{ and } \texttt{False} \not<: \tau_1) \\ \Gamma, \psi_1^+ \overset{\pi}{\vdash} e_2 : \tau;\ \psi_2^+ \mid \psi_2^- \\ \Gamma, \psi_1^- \overset{\pi}{\vdash} e_3 : \tau;\ \psi_3^+ \mid \psi_3^-\end{array}}{\Gamma \overset{\pi}{\vdash} \texttt{if}\, e_1\, e_2\, e_3 : \tau;\ \psi_2^+ \vee \psi_3^+ \mid \psi_2^- \vee \psi_3^-}$$

$$\textsc{T-If-Sym}^{\bullet}\ \frac{\begin{array}{c}\Gamma \overset{\bullet}{\vdash} e_1 : \widehat{\tau}_1;\ \psi_1^+ \mid \psi_1^- \\ \textit{symbolic?}\,\tau_1 \text{ and } \texttt{False} <: \tau_1 \\ \Gamma, \psi_1^+ \overset{\circ}{\vdash} e_2 : \tau;\ \psi_2^+ \mid \psi_2^- \\ \Gamma, \psi_1^- \overset{\circ}{\vdash} e_3 : \tau;\ \psi_3^+ \mid \psi_3^-\end{array}}{\Gamma \overset{\bullet}{\vdash} \texttt{if}\, e_1\, e_2\, e_3 : \widehat{\tau};\ \psi_2^+ \vee \psi_3^+ \mid \psi_2^- \vee \psi_3^-}$$

$$\textsc{T-If-Sym}^{\circ}\ \frac{\begin{array}{c}\Gamma \overset{\circ}{\vdash} e_1 : \widehat{\tau}_1;\ \psi_1^+ \mid \psi_1^- \\ \textit{symbolic?}\,\tau_1 \text{ and } \texttt{False} <: \tau_1 \\ \Gamma, \psi_1^+ \overset{\circ}{\vdash} e_2 : \tau;\ \psi_2^+ \mid \psi_2^- \\ \Gamma, \psi_1^- \overset{\circ}{\vdash} e_3 : \tau;\ \psi_3^+ \mid \psi_3^-\end{array}}{\Gamma \overset{\circ}{\vdash} \texttt{if}\, e_1\, e_2\, e_3 : \widehat{\tau};\ \psi_2^+ \vee \psi_3^+ \mid \psi_2^- \vee \psi_3^-}$$

Fig. 13. Type rules for $\widehat{\lambda_{\bullet}}$ that consider path concreteness ($\overset{\bullet}{\vdash}$: concrete path, $\overset{\circ}{\vdash}$: symbolic path, $\overset{\pi}{\vdash}$: either).

3.5 Concreteness Polymorphism of Paths

Figure 12 shows an extended function type that includes a path concreteness marker where $\bullet$ means concrete path and $\circ$ means symbolic path. More precisely, a function with type marked with $\bullet$ may only be applied in the context of a concrete path whereas a function with type marked with $\circ$ may be applied in any context. Consequently, we split $\widehat{\lambda_{\bullet}}$'s type rules into two sets of judgements, with one set using a $\overset{\bullet}{\vdash}$ relation for type checking under a concrete path, and one set using a $\overset{\circ}{\vdash}$ relation for type checking under a symbolic path. Most of the $\overset{\bullet}{\vdash}$ concrete path rules and $\overset{\circ}{\vdash}$ symbolic path rules are identical to their counterparts from figures 3, 8, and 9 except for the lambda, function application, and conditional rules. Figure 13 shows new versions these rules.[8]

[8]To simplify the presentation, figure 13 omits the o component of its judgements rules since its behavior remains unchanged.

$$\textsc{T-Set!}^{\bullet}\ \frac{x{:}\tau \in \Gamma \qquad \Gamma \vdash^{\bullet} e \ :\ \tau}{\Gamma \vdash^{\bullet} \texttt{set!}\, x\, e \ :\ \texttt{Unit};\ \top \mid \bot} \qquad \textsc{T-Set!}^{\circ}\ \frac{x{:}\widehat{\tau} \in \Gamma \qquad \Gamma \vdash^{\circ} e \ :\ \widehat{\tau}}{\Gamma \vdash^{\circ} \texttt{set!}\, x\, e \ :\ \texttt{Unit};\ \top \mid \bot}$$

$$\textsc{T-Seq}^{\pi}\ \frac{\Gamma \vdash^{\pi} e_1 \ :\ \tau_1 \qquad \Gamma \vdash^{\pi} e_2 \ :\ \tau_2;\ \psi_2^{+} \mid \psi_2^{-}}{\Gamma \vdash^{\pi} e_1 ; e_2 \ :\ \tau_2;\ \psi_2^{+} \mid \psi_2^{-}}$$

Fig. 14. $\widehat{\lambda_{\bullet}}$ type rules for mutation, considering path concreteness.

Whereas the T-Lam-ConcArg and T-Lam-SymArg rules in figure 9 assign lambdas two possible arrow types, one for a symbolic input and one for a concrete one, the T-Lam-ConcArgPath$^{\bullet}$, T-Lam-ConcArg-SymPath$^{\bullet}$, T-Lam-SymArg-ConcPath$^{\bullet}$, and T-Lam-SymArgPath$^{\bullet}$ rules in figure 13 assign lambdas four possible arrow types, where the concreteness of the path is now considered in addition to the concreteness of the input. Specifically, a lambda body is type checked in four different contexts, one corresponding to each kind of input and path combination. Similarly, the T-Lam-ConcArgPath$^{\circ}$, T-Lam-ConcArg-SymPath$^{\circ}$, T-Lam-SymArg-ConcPath$^{\circ}$, and T-Lam-SymArgPath$^{\circ}$ rules assign types to lambdas defined under a symbolic path. Subsequently, the T-App rules enforce that functions are only applied in appropriate contexts. In figure 13, T-App$^{\pi}$ represents two rules where π may be replaced with either $\bullet$ or $\circ$, with the former requiring $\bullet$-marked functions in $\bullet$ paths, and the latter requiring $\circ$-marked functions in $\circ$ paths.

The last rules in figure 13 are the conditional rules. The rules mostly propagate the path concreteness in which the conditional expression appears to the conditional branches, except for T-If-Sym$^{\bullet}$, which switches from $\vdash^{\bullet}$ to $\vdash^{\circ}$ when type checking the branches to properly handle the symbolic path.

With the rules in figure 13, we may define sound mutation rules, shown in figure 14. T-Set!$^{\bullet}$ specifies that any mutation is acceptable in a concrete path while T-Set!$^{\circ}$ allows mutation of only variables with symbolic type. Observe that these rules do not use type judgements in the premise to determine the acceptable type for the new value e, since subtyping would allow lifting the type to be symbolic. Instead, the proposition environment must directly include a proposition that x may have the specified type. Finally, T-Seq$^{\pi}$ specifies that a sequence expression has the type and propositions of its last expression, in both concrete and symbolic paths.

4 METATHEORY

This section describes some desirable properties of $\widehat{\lambda_{\bullet}}$. The key theorem is a soundness result demonstrating that symbolic values cannot cause evaluation to get stuck by flowing to positions that cannot handle them.

4.1 Dynamic Semantics

$$v \in \mathit{Val} ::= i \mid s \mid \texttt{true} \mid \texttt{false} \mid () \mid \mathit{op} \mid \lambda x{:}\tau\,.\,e \mid \widehat{v} \qquad \text{(values)}$$

$$\widehat{v} \in \mathit{SVal} ::= \widehat{x}^{\texttt{Bool}} \mid \widehat{x}^{\texttt{Int}} \mid \widehat{\mathit{op}}\ \widehat{v} \mid \langle [\widehat{v}\colon v] \ldots \rangle \qquad \text{(symbolic values)}$$

Fig. 15. $\widehat{\lambda_{\bullet}}$ values

Figure 15 extends the previous grammars with the notion of values, both symbolic and concrete, which are the result of symbolic execution. Specifically, a value v is either a concrete integer, string, boolean, void value, lambda, primitive function, or a symbolic value $\widehat{v}$. The symbolic values are the base symbolic booleans and integers; symbolic expressions, which result from evaluating

$$M \in Mach ::= \langle e, \rho, \sigma, \kappa \rangle \qquad V \in VMach ::= \langle v, \rho, \sigma, \langle\rangle \rangle \qquad \text{(machine states, final states)}$$
$$\rho \in VEnv ::= x \mapsto \ell, \ldots \qquad \ell \in Loc \qquad \sigma \in Sto ::= \ell \mapsto c, \ldots \qquad c \in Clo ::= \langle v, \rho \rangle \quad \text{(envs, stores)}$$
$$\kappa \in Frames ::= \langle\rangle \mid \left\langle \widehat{\texttt{if}}_1\texttt{:}\ \widehat{v}, \rho_{\widehat{v}}, e, \rho, \sigma, \kappa \right\rangle^{\circ} \mid \left\langle \widehat{\texttt{if}}_2\texttt{:}\ \widehat{v}, \rho_{\widehat{v}}, v, \rho, \sigma, \kappa \right\rangle^{\circ} \mid \qquad \text{(stack frames)}$$
$$\langle \texttt{if:}\ e, e, \rho, \kappa \rangle^{\pi} \mid \langle \texttt{arg:}\ e, \rho, \kappa \rangle^{\pi} \mid \langle \texttt{fn:}\ v, \rho, \kappa \rangle^{\pi} \mid \langle \texttt{set!:}\ \ell, \kappa \rangle^{\pi} \mid \langle \texttt{seq:}\ e, \rho, \kappa \rangle^{\pi}$$

Fig. 16. Grammar for CESK machine semantics for $\widehat{\lambda}_{\bullet}$.

primitive operations applied to symbolic values; or guarded symbolic union values, which result from evaluating symbolic conditionals.

Figure 17 shows how programs evaluate to values. Specifically, using the syntax in figure 16, it defines a CESK-style [Felleisen et al. 2009], register-machine semantics[9] for $\widehat{\lambda}_{\bullet}$. A machine state is a 4-tuple consisting of a (C)ontrol (or (C)urrent) expression e, a value (E)nvironment ρ whose domain includes free variables in e, a (S)tore σ, and a stac(K) κ. Briefly, value environments map variables to store locations, stores map locations to closures c, closures are a value-environment pair, and stack frames represent the context of evaluation.

The machine syntax is mostly standard, except for the stack frames which are decorated with a path concreteness marker, analogous to the type rules in figure 13. The marker determines the concreteness of the path for the subexpressions in the stack frame. These markers do not affect evaluation of programs, however, i.e., no left-hand sides in figure 17 use this path concreteness information. Instead, they are included only to help with the soundness proof. For this reason, figure 17 omits the path markers; instead, we assume a new stack frame implicitly inherits the same marker as its preceding frame, except for $\widehat{\texttt{if}}$ frames, which always have a symbolic path marker.

The rules in figure 17 are also mostly straightforward: evaluation of subexpressions proceeds in call-by-value order, using the stack to save its context. For example APP-FN begins evaluation of an application expression by setting the control expression to be the function and saves the argument expression and a copy of the environment in a new `arg` stack frame.

Figure 17's rules deviate from traditional CESK rules where evaluation mixes both concrete and symbolic values. For example, evaluation of conditionals may produce symbolic values, as illustrated by the $\widehat{\text{IF}}$, $\widehat{\text{IF}}$-THEN, and $\widehat{\text{IF}}$-ELSE rules. Specifically, when the test expression is a symbolic boolean $\widehat{v}$, *both* branches are evaluated and the resulting values, guarded by $\widehat{v}$, make up the branches of the resulting symbolic union value. Evaluation of each branch should occur independently of the other branch, i.e., they should evaluate with different σ stores. Thus the $\widehat{\texttt{if}}_1$ and $\widehat{\texttt{if}}_2$ stack frames save a store σ to accompany the "else" and "then" branches, respectively.

After both branches are evaluated, the $\widehat{\text{IF}}$-ELSE rule uses the $\mu_{\widehat{v}}$ merging function, defined in figure 18, to create a guarded symbolic union value. The function creates the symbolic value from the test value and branch results, taking special care to merge their environments to avoid name conflicts. In addition, $\widehat{\text{IF}}$-ELSE merges the stores from each branch using the μ_{σ} function. The function creates guarded symbolic union values for any locations that point to different values.

One other key rule is APP-OP; it specifies how primitive operations are evaluated using a δ metafunction, whose definition is presented in figure 19. The δ function evaluates application of primitive operations to *concrete* values in the expected manner. In addition, it dictates which and

[9] Our semantics is more or less equivalent to that of Torlak and Bodik [2014] with three key differences: (1) our merging function $\mu_{\widehat{v}}$ is simpler since it is not our main focus; (2) we exclude the solver API and do not include an explicit path register; for our purposes, it is sufficient for individual symbolic values to track their own guard conditions; and (3) we include an explicit value environment, which helps define machine state type judgements in order to show soundness.

$$\langle x, \rho, \sigma, \kappa\rangle \mapsto \langle v, \rho_v, \sigma, \kappa\rangle \quad \text{where } \sigma[\rho[x]] = \langle v, \rho_v\rangle \qquad \text{(VAR)}$$

$$\langle e_1\, e_2, \rho, \sigma, \kappa\rangle \mapsto \langle e_1, \rho, \sigma, \langle \texttt{arg:}\ e_2, \rho, \kappa\rangle\rangle \qquad \text{(APP-FN)}$$

$$\langle v, \rho_v, \sigma, \langle \texttt{arg:}\ e, \rho, \kappa\rangle\rangle \mapsto \langle e, \rho, \sigma, \langle \texttt{fn:}\ v, \rho_v, \kappa\rangle\rangle \qquad \text{(APP-ARG)}$$

$$\left\langle v, \rho, \sigma, \left\langle \texttt{fn:}\ op, \rho_{op}, \kappa\right\rangle\right\rangle \mapsto \langle v', \rho, \sigma, \kappa\rangle \quad \text{where } \delta\, op\, v = v' \qquad \text{(APP-OP)}$$

$$\langle v, \rho_v, \sigma, \langle \texttt{fn:}\ \lambda x\!:\!\tau\,.\,e, \rho, \kappa\rangle\rangle \mapsto \langle e, \rho[x \mapsto \ell], \sigma[\ell \mapsto \langle v, \rho_v\rangle], \kappa\rangle \quad \text{where } \ell \notin dom(\sigma) \qquad \text{(APP-}\beta\text{)}$$

$$\langle \texttt{if}\ e_1\, e_2\, e_3, \rho, \sigma, \kappa\rangle \mapsto \langle e_1, \rho, \sigma, \langle \texttt{if:}\ e_2, e_3, \rho, \kappa\rangle\rangle \qquad \text{(IF)}$$

$$\langle \texttt{false}, \rho_v, \sigma, \langle \texttt{if:}\ e_{then}, e_{else}, \rho, \kappa\rangle\rangle \mapsto \langle e_{else}, \rho, \sigma, \kappa\rangle \qquad \text{(IF-FALSE)}$$

$$\langle v, \rho_v, \sigma, \langle \texttt{if:}\ e_{then}, e_{else}, \rho, \kappa\rangle\rangle \mapsto \langle e_{then}, \rho, \sigma, \kappa\rangle \quad \text{where } v \neq \texttt{false}, v \neq \widehat{v} \qquad \text{(IF-TRUE1)}$$

$$\langle \widehat{v}, \rho_v, \sigma, \langle \texttt{if:}\ e_{then}, e_{else}, \rho, \kappa\rangle\rangle \mapsto \langle e_{then}, \rho, \sigma, \kappa\rangle \quad \text{where } \texttt{bool?}\widehat{v} = \texttt{false} \qquad \text{(IF-TRUE2)}$$

$$\langle \widehat{v}, \rho, \sigma, \langle \texttt{if:}\ e_1, e_2, \rho_1, \kappa\rangle\rangle \mapsto \left\langle e_1, \rho_1, \sigma, \left\langle \widehat{\texttt{if}}_1\texttt{:}\ \widehat{v}, \rho, e_2, \rho_1, \sigma, \kappa\right\rangle\right\rangle \qquad (\widehat{\text{IF}})$$

$$\left\langle v, \rho, \sigma, \left\langle \widehat{\texttt{if}}_1\texttt{:}\ \widehat{v}, \rho_{\widehat{v}}, e_2, \rho_2, \sigma_2, \kappa\right\rangle\right\rangle \mapsto \left\langle e_2, \rho_2, \sigma_2, \left\langle \widehat{\texttt{if}}_2\texttt{:}\ \widehat{v}, \rho_{\widehat{v}}, v, \rho, \sigma, \kappa\right\rangle\right\rangle \qquad (\widehat{\text{IF}}\text{-THEN})$$

$$\left\langle v_2, \rho_2, \sigma_2, \left\langle \widehat{\texttt{if}}_2\texttt{:}\ \widehat{v}, \rho, v_1, \rho_1, \sigma_1, \kappa\right\rangle\right\rangle \mapsto \langle \widehat{v}_3, \rho_3, \mu_\sigma(\langle \widehat{v}, \rho\rangle, \sigma_1, \sigma_2), \kappa\rangle \quad \langle \widehat{v}_3, \rho_3\rangle = \mu_{\widehat{v}}(\langle \widehat{v}, \rho\rangle, \langle v_1, \rho_1\rangle, \langle v_2, \rho_2\rangle) \qquad (\widehat{\text{IF}}\text{-ELSE})$$

$$\langle \texttt{set!}\ x\, e, \rho, \sigma, \kappa\rangle \mapsto \langle e, \rho, \sigma, \langle \texttt{set!:}\ \rho[x], \kappa\rangle\rangle \qquad \text{(SET!)}$$

$$\langle v, \rho, \sigma, \langle \texttt{set!:}\ \ell, \kappa\rangle\rangle \mapsto \langle (), \rho, \sigma[\ell \mapsto \langle v, \rho\rangle], \kappa\rangle \qquad \text{(SET-IT!)}$$

$$\langle e_1\,;e_2, \rho, \sigma, \kappa\rangle \mapsto \langle e_1, \rho, \sigma, \langle \texttt{seq:}\ e_2, \rho, \kappa\rangle\rangle \qquad \text{(SEQ1)}$$

$$\langle v, \rho_v, \sigma, \langle \texttt{seq:}\ e, \rho, \kappa\rangle\rangle \mapsto \langle e, \rho, \sigma, \kappa\rangle \qquad \text{(SEQ2)}$$

Fig. 17. $\widehat{\lambda}_{\bullet}$ CESK machine semantics

how primitive operations should handle *symbolic* values. Some operations, e.g., `strlen` and `str?`, do not support symbolic values and evaluation gets stuck if these operations are applied to symbolic values. In other cases, such as applying `bool?` to $\widehat{x}^{\texttt{Bool}}$,a symbolic input may turn into a concrete value result. More likely, however, applying a primitive operation to a symbolic value results in

$$\boxed{\theta : Clo, VEnv, VEnv \to Clo}$$

$$\theta(\langle v, \rho\rangle, \rho_1, \rho_2) = \langle v[x := y] \ldots, \{y \mapsto \rho[x] \mid x \in dom(\rho), y \notin dom(\rho_1, \rho_2)\}\rangle$$

$$\boxed{\mu_{\widehat{v}} : Clo, Clo, Clo \to Clo}$$

$$\mu_{\widehat{v}}(\langle \widehat{v}, \rho\rangle, \langle v_1, \rho_1\rangle, \langle v_2, \rho_2\rangle) = \left\langle \langle [\widehat{v}_3\colon v_4]\widehat{[\mathsf{not}\ \widehat{v}_3\colon v_5]}\rangle, \rho_3 \cup \rho_4 \cup \rho_5 \right\rangle$$
$$\text{where } \langle \widehat{v}_3, \rho_3\rangle = \theta(\langle \widehat{v}, \rho\rangle, \rho_1, \rho_2)$$
$$\langle v_4, \rho_4\rangle = \theta(\langle v_1, \rho_1\rangle, \rho, \rho_2)$$
$$\langle v_5, \rho_5\rangle = \theta(\langle v_2, \rho_2\rangle, \rho, \rho_1)$$

$$\boxed{\mu_\sigma : Clo, Sto, Sto \to Sto}$$

$$\begin{aligned}\mu_\sigma(\langle \widehat{v}, \rho\rangle, \sigma_1, \sigma_2) = {} & \{\ell \mapsto \sigma_1[\ell] \mid \ell \in dom(\sigma_1), \ell \notin dom(\sigma_2)\} \cup \\ & \{\ell \mapsto \sigma_2[\ell] \mid \ell \in dom(\sigma_2), \ell \notin dom(\sigma_1)\} \cup \\ & \{\ell \mapsto \sigma_1[\ell] \mid \ell \in dom(\rho_1), \ell \in dom(\rho_2), \sigma_1[\ell] = \sigma_2[\ell]\} \cup \\ & \{\ell \mapsto \mu_{\widehat{v}}(\langle \widehat{v}, \rho\rangle, \sigma_1[\ell], \sigma_2[\ell]) \mid \ell \in dom(\rho_1), \ell \in dom(\rho_2), \sigma_1[\ell] \neq \sigma_2[\ell]\}\end{aligned}$$

Fig. 18. Merge function for environments and stores.

$$\begin{aligned}
\delta\,\mathsf{conc?}\,\widehat{v} &= \mathsf{false} & \delta\,\mathsf{conc?}\,v &= \mathsf{true},\ \text{where } v \neq \widehat{v} \\
\delta\,\mathsf{bool?}\,\mathsf{true} &= \mathsf{true} & \delta\,\mathsf{str?}\,s &= \mathsf{true} \\
\delta\,\mathsf{bool?}\,\mathsf{false} &= \mathsf{true} & \delta\,\mathsf{str?}\,v &= \mathsf{false},\ \text{where } v \neq s, v \neq \widehat{v} \\
\delta\,\mathsf{bool?}\,v &= \mathsf{false} & \delta\,\mathsf{strlen}\,s &= \#\text{ chars in } s \\
\text{where } v \neq \mathsf{true},\ v \neq \mathsf{false}, v \neq \widehat{v} & & \delta\,\mathsf{int?}\,i &= \mathsf{true} \\
\delta\,\mathsf{bool?}\,\widehat{x}^{\mathsf{Bool}} &= \mathsf{true} & \delta\,\mathsf{int?}\,v &= \mathsf{false}, v \neq i, v \neq \widehat{v} \\
\delta\,\mathsf{bool?}\,\widehat{x}^{\tau} &= \mathsf{false},\ \text{where } \tau \neq \mathsf{Bool} & \delta\,\mathsf{int?}\,\widehat{x}^{\mathsf{Int}} &= \mathsf{true} \\
\delta\,\mathsf{bool?}\,\widehat{(\mathsf{not}\ \widehat{v})} &= \mathsf{true} & \delta\,\mathsf{int?}\,\widehat{x}^{\tau} &= \mathsf{false},\ \text{where } \tau \neq \mathsf{Int} \\
\delta\,\mathsf{bool?}\,(\widehat{op}\ \widehat{v}) &= \mathsf{false},\ \text{where } op \neq \mathsf{not} & \delta\,\mathsf{int?}\,\widehat{(\mathsf{add1}\ \widehat{v})} &= \mathsf{true} \\
\delta\,\mathsf{bool?}\,\langle[\widehat{v}\colon v]\ldots\rangle &= \langle[\widehat{v}\colon \delta\,\mathsf{bool?}\,v]\ldots\rangle & \delta\,\mathsf{int?}\,(\widehat{op}\ \widehat{v}) &= \mathsf{false},\ \text{where } op \neq \mathsf{add1} \\
\delta\,\mathsf{not}\,\mathsf{false} &= \mathsf{true} & \delta\,\mathsf{int?}\,\langle[\widehat{v}\colon v]\ldots\rangle &= \langle[\widehat{v}\colon \delta\,\mathsf{int?}\,v]\ldots\rangle \\
\delta\,\mathsf{not}\,v &= \mathsf{false},\ v \neq \widehat{v}, v \neq \mathsf{false} & \delta\,\mathsf{add1}\,i &= i + 1 \\
\delta\,\mathsf{not}\,\widehat{x}^{\mathsf{Bool}} &= \widehat{\mathsf{not}\ \widehat{x}^{\mathsf{Bool}}} & \delta\,\mathsf{add1}\,\widehat{x}^{\mathsf{Int}} &= \widehat{\mathsf{add1}\ \widehat{x}^{\mathsf{Int}}} \\
\delta\,\mathsf{not}\,\widehat{x}^{\tau} &= \mathsf{true},\ \text{where } \tau \neq \mathsf{Bool} & \delta\,\mathsf{add1}\,\langle[\widehat{v}\colon v]\ldots\rangle &= \langle[\widehat{v}\colon \delta\,\mathsf{add1}\,v']\ldots\rangle \\
\delta\,\mathsf{not}\,\langle[\widehat{v}\colon v]\ldots\rangle &= \langle[\widehat{v}\colon \delta\,\mathsf{not}\,v]\ldots\rangle & \text{where } v' \in v\ldots, \mathsf{int?}\,v' &
\end{aligned}$$

Fig. 19. Evaluation of $\widehat{\lambda_{\bullet}}$ primitive operations.

$$\textsc{T-Op-SymVal}\ \frac{\Gamma \Vdash^{\pi} op\,\widehat{v} \ :\ \tau;\ \psi^{+} \mid \psi^{-};\ o}{\Gamma \Vdash^{\pi} \widehat{op}\,\widehat{v} \ :\ \tau;\ \psi^{+} \mid \psi^{-};\ o}$$

$$\textsc{T-Union-SymVal}\ \frac{\Gamma \Vdash^{\pi} \widehat{v}_1 \ :\ \tau_1;\ \psi_1^{+} \mid \psi_1^{-};\ o_1 \qquad \Gamma, \psi_1^{+} \Vdash v_1 \ :\ \tau;\ \psi_3^{+} \mid \psi_3^{-};\ o \qquad \Gamma \Vdash^{\pi} \widehat{v}_2 \ :\ \tau_2;\ \psi_2^{+} \mid \psi_2^{-};\ o_2 \qquad \Gamma, \psi_2^{+} \Vdash v_2 \ :\ \tau;\ \psi_4^{+} \mid \psi_4^{-};\ o}{\Gamma \Vdash^{\pi} \langle [\widehat{v}_1 : v_1][\widehat{v}_2 : v_2] \rangle \ :\ \tau;\ \psi_3^{+} \vee \psi_3^{-} \mid \psi_4^{+} \vee \psi_4^{-};\ o}$$

$$\text{TM-CESK}\ \frac{\mathcal{E}(\rho,\sigma) \Vdash^{\pi} e : \tau_2;\ \psi_2^{+} \mid \psi_2^{-};\ o_2 \qquad \tau_2;\ \psi_2^{+} \mid \psi_2^{-};\ o_2; \sigma \vdash_{\kappa} \kappa^{\pi} : \tau;\ \psi^{+} \mid \psi^{-};\ o}{\vdash_{M} \langle e, \rho, \sigma, \kappa^{\pi} \rangle : \tau;\ \psi^{+} \mid \psi^{-};\ o}$$

Fig. 20. Type judgements for machine states and symbolic values.

$$\boxed{\mathcal{E} : VEnv, Sto \rightarrow Env}$$

$$\mathcal{E}(\rho,\sigma) = \{x : \tau \mid x \in dom(\rho)\}$$
$$\text{where } \langle v, \rho_2 \rangle = \sigma[\rho[x]] \text{ and } \mathcal{E}(\rho_2,\sigma) \vdash v \ :\ \tau$$

Fig. 21. Metafunction converting value and environment and store to proposition environment.

another symbolic value. For example, applying add1 to $\widehat{x}^{\mathtt{Int}}$ results in the symbolic value $\widehat{\mathtt{add1}}\ \widehat{x}^{\mathtt{Int}}$. Finally, applying a primitive operation to a guarded union value traverses the tree, recursively applying the primitive operation. The most interesting case is applying add1 to such a guarded union value. In this case, the tree is additionally pruned of non-integer branches.

4.2 Soundness

To evaluate a program e using the semantics in figure 17, an *inject* function first compiles the program to initial machine configuration $\langle e, \cdot, \cdot, \langle\rangle\rangle$ with an empty environment, store, and stack. Using this function, theorem 4.1 states the main theoretical result, which is that evaluating a well-typed program cannot get stuck.

Theorem 4.1 (Soundness). *If* $\vdash e \ :\ \tau;\ \psi^{+} \mid \psi^{-};\ o$, $M = inject(e)$, *and evaluating* M *terminates, then* $M \mapsto^{*} V$, *and* $\vdash_{M} V : \tau;\ \psi^{+\prime} \mid \psi^{-\prime};\ o'$ *for some* $\psi^{+\prime}$, $\psi^{-\prime}$, *and* o'.

We prove theorem 4.1 using a standard progress and preservation approach [Wright and Felleisen 1994]. To do so, we first need to define typing judgements for symbolic values and machine states, both shown in figure 20. The T-Op-SymVal uses the rule for application from figure 13 and the T-Union-SymVal mostly mirrors the type rule for `if`.

The $\vdash_{M}$ relation for machine states is, essentially, a "bottom-up" version of the traditional "top-down" type judgements for expressions from the previous sections, e.g., figure 13. Specifically, $\vdash_{M}$ first uses those expression type judgements to type check the control expression. It then feeds the output of that judgement to a $\vdash_{\kappa}$ judgement, defined in figures 22 and 23, for type checking the stack. Intuitively, this input type information fills the "hole" that is implicit in each stack frame.

The $\vdash_{M}$ and $\vdash_{\kappa}$ relations both rely on a $\mathcal{E}$ metafunction, defined in figure 21, that "uncompiles" a runtime value environment and store into a proposition environment for type checking. This $\mathcal{E}$ metafunction computes types for elements of the value environment using the type judgements from figure 13. This function does not have access to path concreteness information, however, but as lemma 4.2 states, it does not matter which variant of the type judgement is used.

TK-Empty

$$\tau;\ \psi^+ \mid \psi^-;\ o;\ \sigma \vdash_\kappa \langle\rangle : \tau;\ \psi^+ \mid \psi^-;\ o$$

TK-If-Conc$^\pi$

$$\frac{\begin{array}{c}\textit{concrete?}\,\tau_{in} \text{ or } (\textit{symbolic?}\,\tau_{in} \text{ and } \texttt{False} \not<: \tau_{in}) \\ \mathcal{E}(\rho,\sigma), \psi^+_{in} \Vdash^\pi e_1 \ :\ \tau;\ \psi^+_1 \mid \psi^-_1;\ o \\ \mathcal{E}(\rho,\sigma), \psi^-_{in} \Vdash^\pi e_2 \ :\ \tau;\ \psi^+_2 \mid \psi^-_2;\ o \\ \tau;\ \psi^+_1 \vee \psi^+_2 \mid \psi^-_1 \vee \psi^-_2\ o;\ \sigma \vdash_\kappa \kappa : \tau';\ \psi^+ \mid \psi^-;\ o'\end{array}}{\tau_{in};\ \psi^+_{in} \mid \psi^-_{in};\ o_{in};\ \sigma \vdash_\kappa \langle \texttt{if}\colon e_1, e_2, \rho, \kappa\rangle^\pi : \tau';\ \psi^+ \mid \psi^-;\ o'}$$

TK-If-Sym$^\pi$

$$\frac{\begin{array}{c}\textit{symbolic?}\,\tau_{in} \text{ and } \texttt{False} <: \tau_{in} \\ \mathcal{E}(\rho,\sigma), \psi^+_{in} \Vdash^\circ e_1 \ :\ \tau;\ \psi^+_1 \mid \psi^-_1; o \\ \mathcal{E}(\rho,\sigma), \psi^-_{in} \Vdash^\circ e_2 \ :\ \tau;\ \psi^+_2 \mid \psi^-_2; o \\ \widehat{\tau};\ \psi^+_1 \vee \psi^+_2 \mid \psi^-_1 \vee \psi^-_2\ o;\ \sigma \vdash_\kappa \kappa : \tau';\ \psi^+ \mid \psi^-;\ o'\end{array}}{\widehat{\tau}_{in}; \psi^+_{in} \mid \psi^-_{in}; o_{in}; \sigma \vdash_\kappa \langle \texttt{if}\colon e_1, e_2, \rho, \kappa\rangle^\pi : \tau';\ \psi^+ \mid \psi^-;\ o'}$$

TK-SymIf1°

$$\frac{\begin{array}{c}\mathcal{E}(\rho_2,\sigma_2) \Vdash^\circ e_2 \ :\ \tau_1;\ \psi^+_2 \mid \psi^-_2; o_1 \\ \widehat{\tau}_1; \psi^+_1 \vee \psi^+_2 \mid \psi^-_1 \vee \psi^-_2; o_1; \mu_\sigma(\langle \widehat{v}, \rho\rangle, \sigma_1, \sigma_2) \vdash_\kappa \kappa : \tau';\ \psi^+ \mid \psi^-;\ o'\end{array}}{\tau_1;\ \psi^+_1 \mid \psi^-_1;\ o_1;\ \sigma_1 \vdash_\kappa \left\langle \widehat{\texttt{if}}_1\colon \widehat{v}, \rho, e_2, \rho_2, \sigma_2, \kappa\right\rangle^\circ : \tau';\ \psi^+ \mid \psi^-;\ o'}$$

TK-SymIf2°

$$\frac{\begin{array}{c}\mathcal{E}(\rho_1,\sigma_1) \Vdash^\circ e_1 \ :\ \tau_2;\ \psi^+_1 \mid \psi^-_1; o_2 \\ \widehat{\tau}_2; \psi^+_2 \vee \psi^+_1 \mid \psi^-_2 \vee \psi^-_1; o_2; \mu_\sigma(\langle \widehat{v}, \rho\rangle, \sigma_1, \sigma_2) \vdash_\kappa \kappa : \tau';\ \psi^+ \mid \psi^-;\ o'\end{array}}{\tau_2;\ \psi^+_2 \mid \psi^-_2;\ o_2;\ \sigma_2 \vdash_\kappa \left\langle \widehat{\texttt{if}}_2\colon \widehat{v}, \rho, e_1, \rho_1, \sigma_1, \kappa\right\rangle^\circ : \tau';\ \psi^+ \mid \psi^-;\ o'}$$

TK-App1$^\pi$

$$\frac{\begin{array}{c}\mathcal{E}(\rho,\sigma) \Vdash^\pi e \ :\ \tau_1;\ \psi^+_e \mid \psi^-_e;\ o_e \\ \tau_2[x := o_e]; \psi^+[x := o_e] \mid \psi^-[x := o_e]; o[x := o_e]; \sigma \vdash_\kappa \kappa : \tau_3;\ \psi^+_3 \mid \psi^-_3;\ o_3\end{array}}{\pi; x\colon\tau_1 \xrightarrow{\psi^+ \mid \psi^-; o} \tau_2; \psi^+_{in} \mid \psi^-_{in}; o_{in}; \sigma \vdash_\kappa \langle \texttt{arg}\colon e, \rho, \kappa\rangle^\pi : \tau_3;\ \psi^+_3 \mid \psi^-_3;\ o_3}$$

TK-App2$^\pi$

$$\frac{\begin{array}{c}\mathcal{E}(\rho,\sigma) \Vdash^\pi v \ :\ \pi; x\colon\tau_1 \xrightarrow{\psi^+ \mid \psi^-; o} \tau_2;\ \psi^+_v \mid \psi^-_v;\ o_v \\ \tau_2[x := o_{in}]; \psi^+[x := o_{in}] \mid \psi^-[x := o_{in}]; o[x := o_{in}]; \sigma \vdash_\kappa \kappa : \tau_3;\ \psi^+_3 \mid \psi^-_3;\ o_3\end{array}}{\tau_1; \psi^+_{in} \mid \psi^-_{in}; o_{in}; \sigma \vdash_\kappa \langle \texttt{fn}\colon v, \rho, \kappa\rangle^\pi : \tau_3;\ \psi^+_3 \mid \psi^-_3;\ o_3}$$

TK-Seq$^\pi$

$$\frac{\mathcal{E}(\rho,\sigma) \Vdash^\pi e \ :\ \tau_e;\ \psi^+_e \mid \psi^-_e;\ o_e \qquad \tau_e;\ \psi^+_e \mid \psi^-_e;\ o_e; \sigma \vdash_\kappa \kappa : \tau;\ \psi^+ \mid \psi^-;\ o}{\tau_{in};\ \psi^+_{in} \mid \psi^-_{in};\ o_{in};\ \sigma \vdash_\kappa \langle \texttt{seq}\colon e, \rho, \kappa\rangle^\pi : \tau;\ \psi^+ \mid \psi^-;\ o}$$

Fig. 22. Type judgements for stack frames, part 1.

TK-Set!•

$$\frac{\mathcal{E}(\rho,\sigma) \overset{\bullet}{\vdash} v \;:\; \tau_{in};\; \psi_v^+ \mid \psi_v^-;\; o_v \qquad \overset{concrete?\,\tau_{in}}{\text{where } \sigma[\ell] = \langle v, \rho\rangle} \qquad \texttt{Unit}; \top; \bot; \cdot; \sigma \vdash_\kappa \kappa : \tau;\; \psi^+ \mid \psi^-;\; o}{\tau_{in};\; \psi_{in}^+ \mid \psi_{in}^-;\; o_{in};\; \sigma \vdash_\kappa \langle \texttt{set!}\colon \ell, \kappa\rangle^\bullet : \tau;\; \psi^+ \mid \psi^-;\; o}$$

TK-Set!-Sym•

$$\frac{\mathcal{E}(\rho,\sigma) \overset{\bullet}{\vdash} \widehat{v} \;:\; \widehat{\tau}_{in};\; \psi_v^+ \mid \psi_v^-;\; o_v \qquad \text{where } \sigma[\ell] = \langle \widehat{v}, \rho\rangle \qquad \texttt{Unit}; \top; \bot; \cdot; \sigma \vdash_\kappa \kappa : \tau;\; \psi^+ \mid \psi^-;\; o}{\widehat{\tau}_{in}; \psi_{in}^+ \mid \psi_{in}^-; o_{in}; \sigma \vdash_\kappa \langle \texttt{set!}\colon \ell, \kappa\rangle^\bullet : \tau;\; \psi^+ \mid \psi^-;\; o}$$

TK-Set!°

$$\frac{\mathcal{E}(\rho,\sigma) \overset{\circ}{\vdash} v \;:\; \tau_{in};\; \psi_v^+ \mid \psi_v^-; o_v \qquad \text{where } \sigma[\ell] = \langle \widehat{v}, \rho\rangle \qquad \texttt{Unit}; \top; \bot; \cdot; \sigma \vdash_\kappa \kappa : \tau;\; \psi^+ \mid \psi^-;\; o}{\tau_{in}; \psi_{in}^+ \mid \psi_{in}^-; o_{in}; \sigma \vdash_\kappa \langle \texttt{set!}\colon \ell, \kappa\rangle^\circ : \tau;\; \psi^+ \mid \psi^-;\; o}$$

Fig. 23. Type judgements for stack frames, part 2: `set!` frames.

Lemma 4.2 (Types for Values). *For all* v, $\Gamma \overset{\bullet}{\vdash} v \;:\; \tau;\; \psi^+ \mid \psi^-;\; o$ *iff* $\Gamma \overset{\circ}{\vdash} v \;:\; \tau;\; \psi^+ \mid \psi^-;\; o$.

The $\vdash_\kappa$ rules in figure 22 are mostly straightforward, mirroring their counterparts in figure 13. Specifically, the rules for `if` use a symbolic $\overset{\circ}{\vdash}$ if the conditional test is symbolic, and the application rules require a function type with path concreteness that matches the marker on the stack frame. The interesting $\vdash_\kappa$ rules are for `set!` frames, in figure 23. When in a concrete path, TK-Set!• and TK-Set!-Sym• require the concreteness of τ_{in} to match the concreteness of the value already at location ℓ. When the path is symbolic, TK-Set!° specifies that type checking only succeeds if the value at location ℓ is symbolic.

With these definitions, we can state our key lemmas 4.3 and 4.4

Lemma 4.3 (Progress). *If* $\vdash_M M : \tau;\; \psi^+ \mid \psi^-;\; o$ *then either* $M \in VMach$ *or* $\exists M'$ *s.t.* $M \mapsto M'$.

Lemma 4.4 (Preservation). *If* $\vdash_M M : \tau;\; \psi^+ \mid \psi^-;\; o$ *and* $M \mapsto M'$, *then* $\vdash_M M' : \tau;\; \psi^{+\prime} \mid \psi^{-\prime};\; o'$.

Intuitively, lemma 4.3 states that well-typed terms either are values or may make an evaluation step. The proof consists of a case analysis of the various possible combinations of control expressions and topmost stack frames. In particular, observe that in figure 17, when the control string is a value and the stack is non-empty, evaluation only gets stuck when applying a non-function or applying a primitive to the wrong type of value or value concreteness, all of which are type errors.

Lemma 4.4 states that machine transitions preserve well-typedness. The proof again proceeds by a case analysis of each machine transition rule. Interestingly, determining the type of a machine state was more involved than showing type preservation across each step of evaluation. Computing the former required bringing together four aforementioned components: (1) the value environment ρ, from which the type environment is computed, (2) the $\vdash_\kappa$ judgements in figures 22 and 23 that invert the top-down type checking implied by conventional type judgements, thus matching the inverted nature of a stack-based machine state, (3) the concreteness tags on stack frames, which help determine proper type concreteness, and (4) the store merging function μ_σ. In particular, figure 22's TK-SymIf1° and TK-SymIf2° utilize all four components. These rules rely on the concreteness marker on the stack frame to properly convert the type of the conditional into symbolic types, as seen in the second premise of each of these rules (τ_1 and τ_2, respectively). In addition, these rules rely on the μ_σ merging function to combine the two copies of the store σ_1 and σ_2. In cases where the two store copies have conflicting values for the same binding, e.g., in terms like `(if` $\widehat{x}^{\texttt{Bool}}$ `(set! y 1) (set! y 2))`, the merging function properly creates a symbolic value, in this case for `y`.

Correspondingly, the right-hand side of figure 17's $\widehat{\text{IF}}$-else evaluation step uses the same merging function and thus preservation holds for conditional expressions.

Together, the two lemmas prove theorem 4.1 and thus we may conclude that symbolic values cannot cause evaluation to get stuck.

5 TYPED ROSETTE

To show that $\widehat{\lambda}_{\bullet}$ can model a real programming language, we implemented Typed Rosette, which adds our type system to the untyped Rosette language [Torlak and Bodik 2014].

5.1 Implementation Organization

Figure 24 depicts the overall architecture of Rosette and Typed Rosette, which are implemented with Racket. Racket's distinguishing features are its modern macro system [Flatt 2002, 2016]—a descendant of its Lisp and Scheme predecessors—and DSL-building capabilities [Tobin-Hochstadt et al. 2011]. Together these features enable programmers to extend and reuse parts of the Racket's implementation in order to create embedded DSLs quickly and easily, e.g., adding symbolic computation capabilities and a solver interface to create Rosette. Even better, Rosette programmers may continue to utilize Racket's DSL-building features to easily build their own solver-aided DSLs.

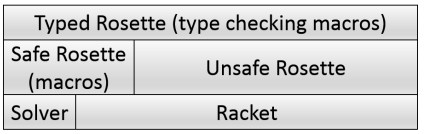

Fig. 24. Typed Rosette Organization

Rosette consists of two sublanguages: a small, safe subset where all language constructs are equipped to handle symbolic values, and a larger language that includes the rest of Racket. The documentation warns programmers to stay within the safe subset but as with all languages, programmers are eventually enticed by the extra features and expressiveness of the "unsafe" parts. Much of Typed Rosette's design focuses on helping the latter group.

5.2 Implementation

Typed Rosette is implemented as an extension of Rosette, also using Racket's macro system. Creating Typed Rosette required implementing a type system, as well as an elaboration pass that translates the typed language to untyped Rosette. Since the elaboration pass occasionally requires type information to direct the elaboration output, e.g., when translating symbolic values like $\widehat{x}^{\texttt{Int}}$ to its untyped representation, we did not implement separate passes. Instead, we used the "type systems as macros" technique of Chang et al. [2017], which allows implementing interleaved type checking and elaboration passes as a series of user-level macro definitions. With this approach is it trivial to implement elaborations that depend on type checking information. It also allowed us to use the existing Rosette implementation without modification. Overall Typed Rosette's implementation added roughly 2500 lines of code to the existing 10,000 lines of Rosette's implementation.

Figure 25 presents the essence of a few type checking macros from Typed Rosette's implementation. They use the `define-typerule` form from Chang et al.'s meta-DSL for creating typed DSLs. The `if` rule on the left consists of two clauses, one each for concrete and symbolic test expressions, respectively, just like the rules from figure 8. The bodies of these `if` clauses also resemble their mathematical counterparts because it uses a bidirectional [Pierce and Turner 1998]-style syntax that interleaves type checking and elaboration. Specifically, a judgement $\vdash e \gg \overline{e}\ (\Rightarrow k\ v)\ldots$ should be read "e elaborates to $\overline{e}$ and produces values $v\ldots$ associated with keys $k\ldots$." For example, the first premise in the first `if` clause elaborates the test expression e_1 to $\overline{e}_1$, and in the process computes its type τ, keyed with symbol `:`, as well as propositions ψ^+ and ψ^-. The next line guards the clause with a check of whether the test expression's type is symbolic. If the check fails, type

```
(define-typerule if
 [(if e₁ e₂ e₃) ≫ ; concrete case
   [⊢ e₁ ≫ ē₁ (⇒ : τ) (⇒ ψ⁺ ψ⁺) (⇒ ψ⁻ ψ⁻)]
   #:unless (and (symbolic? τ) (typecheck? τ False))
   [⊢ (with-prop e₂ ψ⁺) ≫ ē₂ ⇒ τ₁]
   [⊢ (with-prop e₃ ψ⁻) ≫ ē₃ ⇒ τ₂]
   ----------------------------------------
   [⊢ (rosette:if ē₁ ē₂ ē₃) ⇒ (⊔ τ₁ τ₂)]]
 [(if e₁ e₂ e₃) ≫ ; symbolic case
   [⊢ e₁ ≫ ē₁ (⇒ : τ) (⇒ ψ⁺ ψ⁺) (⇒ ψ⁻ ψ⁻)]
   [⊢ (with-prop e₂ ψ⁺) ≫ ē₂ ⇒ τ₁ #:mode sym-path]
   [⊢ (with-prop e₃ ψ⁻) ≫ ē₃ ⇒ τ₂ #:mode sym-path]
   ----------------------------------------
   [⊢ (rosette:if ē₁ ē₂ ē₃) ⇒ (⊔~ τ₁ τ₂)]])
```

```
(define-typerule set!
 [(set! x e) ≫ #:when (sym-path?)
   [⊢ x ≫ x̄ ⇒ τ]
   #:fail-unless (symbolic? τ)
     "sym path requires sym type"
   [⊢ e ≫ ē ⇐ τ]
   ------------------------------
   [⊢ (rosette:set! x̄ ē) ⇒ Unit])
 [(set! x e) ≫ #:when (conc-path?)
   [⊢ x ≫ x̄ ⇒ τ]
   [⊢ e ≫ ē ⇐ τ]
   ------------------------------
   [⊢ (rosette:set! x̄ ē) ⇒ Unit])
```

Fig. 25. A few typechecking macros from Typed Rosette's implementation

checking falls through to the second clause. The next two premises type check the branches using the relevant propositions, and also toggle a flag to indicate a symbolic path. Finally, the output expression and its associated type lie below the conclusion line. Specifically, Typed Rosette's `if` elaborates to untyped Rosette's `if`. The output type is a join of the types of the two branches, computed with ⊔, and the second clause further turns its output type into a symbolic type. The second clause also sets a `sym-path` "mode", which is essentially a flag that holds while typechecking the branches.

The `set!` rule (figure 25, right) also consists of two clauses, distinguished with `sym-path?` and `conc-path?` predicates, which utilize the "mode" flag set by `if`. Thus the rule requires that in a symbolic path, the target of mutation must have symbolic type.

5.3 Concreteness Polymorphism in Practice

Typed Rosette extends $\widehat{\lambda}_{\bullet}$ with additional features found in practical languages such as optional arguments and variable-arity polymorphism [Strickland et al. 2009]. Unlike $\widehat{\lambda}_{\bullet}$, where lambdas are always assigned types with all combinations of concrete/symbolic arguments, Typed Rosette programmers can control which combinations are included in a function's type using signature declarations. For example, the concreteness of this function's output matches its first input:

```
#lang typed/rosette
(: add : (case-> (-> Int Int Int)
                 (-> Int~ Int Int~)))
(define (add x y) (+ x y))
```

The `case->` constructor resembles the intersection type constructor from $\widehat{\lambda}_{\bullet}$, except each constituent function type is considered in the listed order when the function is applied. The signature additionally specifies that the second input must always be concrete and does not affect the output type. Here is another example, where the second argument is now optional:

```
(: add/opt : (case-> (->* [Int] [Int] Int)
                     (->* [Int~] [Int] Int~)))
(define (add/opt x [y 0]) (+ x y))
```

The `->*` function type constructor requires three arguments: a list of types for required arguments, a list of types for optional arguments, and an output type. In both these cases, functions are implicitly assigned types considering both a symbolic and concrete path, as in $\widehat{\lambda}_{\bullet}$.

5.4 Handling Imprecision: Design Discussion

Concreteness polymorphism helps to increase precision but in some cases, Typed Rosette still rejects valid programs. Specifically, figure 18's $\mu_{\widehat{v}}$ merging function highlights where the type system and evaluation may diverge. While $\widehat{\lambda}_{\bullet}$'s merging function always creates a symbolic value, an actual language can more aggressively preserve concrete values. For example, the value $\langle[\widehat{x}^{\texttt{Bool}}: 1][\neg\widehat{x}^{\texttt{Bool}}: 1]\rangle$ could be replaced with just 1. One of Rosette's key innovations is a novel merging algorithm that reduces the number of symbolic values during evaluation. For example, Typed Rosette rejects the following valid Rosette program, which uses the `add` defined above:

```
(define-symbolic* b : Bool)
(add 1 (if b 2 2)) ; => TYERR: expected Int given ^Int
```

The function does not type check because `add`'s type requires a concrete second argument but the type system cannot prove this. To satisfy the type checker, we must use occurrence typing:

```
(let ([x (if b 2 2)])
  (if (conc? x)
      (add 1 x) ; => 3
      (error "expected concrete val")))
```

Another source of imprecision comes from Rosette's pruning of infeasible paths. For example:

```
#lang rosette
(define-symbolic* b boolean?)
(+ 1 (if b 2 "bad")) ; => 3
```

```
#lang typed/rosette
(define-symbolic* b boolean?)
(+ 1 (assert-type (if b 2 "bad") : Int))
; => 3, with assertion b = true
```

The left program succeeds in Rosette but Typed Rosette rejects the program because the second argument's union type is incompatible with addition. Even though the left program is valid, we ultimately chose to reject such programs because they do not occur too frequently in practice. Further, supporting such programs would impose a large cost on the type system implementation, since it would require duplicating large parts of Rosette's runtime path pruning and merging during type checking. Programmers who truly wish to utilize this behavior may either use occurrence typing, or may add an `assert-type` annotation (seen on the right), which restricts the type of a value but generates an additional assertion for the solver must satisfy.

5.5 Unsupported Features

Although unsafe Rosette exposes all of Racket's features to programmers, Typed Rosette's support of Racket is usable but far from complete. For example, Typed Rosette does not support particularly dynamic features such as `eval`. When encountering such features, programmers can separate the untyped code with a typed "shim" layer that typically looks something like:

```
; this file named typed-lib; typed code needing untyped-lib now imports typed-lib instead
(require untyped-lib) ; contains untyped function f
(provide (typed-out [f : ty])) ; assign type ty to f
```

6 EVALUATION

To determine whether Typed Rosette is useful, we ported a large code base from untyped Rosette, summarized in table 1. We sought to confirm that Typed Rosette (1) sufficiently accommodates Rosette idioms and (2) helps with debugging lenient symbolic execution. Our test suite consists of two parts. The first part (table 1, row 1) consists of small examples mainly drawn from Rosette's documentation. The second part (in the other table rows) involves real Rosette applications.

Table 1. Summary of programs ported to Typed Rosette

Name	Description	Untyped LoC	+Typed LoC	Typed Tests	Casts
tests	coverage tests, corner cases			1884	15
fsm	debugging automata	162	+86	54	
bv	synth loop-free bitvector progs	434	+101	438	
ifc	verify non-interference	962	+137	517	
synthcl	synth/verify opencl progs	2632	+615	1609	
ocelot	relational logic library	1757	+396	439	8
inc	synth incremental algs	5445	+634	1134	12

The first part amounted to approximately 2000 lines of "coverage tests." While most used only safe Rosette, a few demonstrated errors from naive use of unsafe constructs. Typed Rosette was able to catch all these latter cases. The last column of table 1 shows that we needed 15 casts to successfully type check the small test cases. Since the documentation includes many corner cases, however, with most involving Rosette's infeasible path pruning as described in section 5.4, it's not too surprising that we needed to help the type checker in these cases.

The second part of our evaluation sought to determine whether Typed Rosette is useful in practice. Specifically, we ported a series of solver-aided DSLs (SDSLs) and example programs from Rosette to Typed Rosette. Table 1 summarizes these efforts: the third column lists the size of the untyped code base; the fourth column shows how many lines were needed to add types; the fifth column shows how many lines of tests we ported to the typed version; and the last column shows how many casts we needed, if any. While porting functions was mostly straightforward, some projects defined their own language extensions, which required more effort since it involves adding new type rules to our type system. For the latter case, we often had to reimplement parts those features using the `define-typerule` described in section 5.2. This partly explains why some line counts in the fourth column may be higher than others. Also, we counted function type signatures with normal typed code in column four, but other needed annotations were counted in column six, e.g., see section 6.3. The rest of this section highlights a few cases in more detail.

6.1 Synthesizing Loop-Free Bitvector Programs

This SDSL helps programmers synthesize bitvector programs as described in Gulwani et al. [2011]. Some language positions do not allow symbolic values and types help identify any violations:

```
#lang typed/bv
(bv 1 4) ; constructs length 4 bitvector with value 1
(define-symbolic* x : Int)
(bv x 4) ; => TYERR: expected Int, got x with type Int
```

Since the language is small and fairly mature, however, we found no errors due to misused symbolic values. Further, this language illustrates how Rosette programmers often manage interaction of symbolic values and unlifted constructs—by creating additional syntactic abstractions, e.g.:

```
(define-fragment (mk-trailing-0s-mask/synth x)
  #:library (bvlib [(bv 1 4) bvsub bvand bvnot 1])
  #:implements mk-trailing-0s-mask)
```

This code tries to synthesize a `mk-trailing-0s-mask/synth` function from a list of "components" and a reference function. Specifically, `define-fragment` encodes the given components as symbolic values representing holes in a static single-assignment program, and then attempts to compute a satisfying assignment for those holes. The abstraction prevents programmers from interacting with

the generated symbolic values, thus reducing the chance of errors. Such forms, however, cannot enforce whether they receive symbolic values or not, and thus our types enhance such abstractions.

6.2 A Library for Relational Logic Specifications

Despite the use of syntactic abstraction to minimize unsafe interactions, symbolic values can still reach unlifted positions. This section reports on an example we encountered while porting the Ocelot library,[10] which extends Rosette with the ability to write, verify, and synthesize Alloy-like [Jackson 2002] relational specifications. Consider this example from the Ocelot documentation:

```
#lang rosette
(define Un (universe '(a b c d))) ; declare universe of atoms
(define cats (declare-relation 1 "cats")) ; declare a "cats" relation
(define iCats (instantiate-bounds ; create an interpretation for "cats"
               (bounds Un (list (make-upper-bound cats '((a) (b) (c) (d)))))))
(define F (&& (some cats) (some (- univ cats)))) ; find an interesting model for "cats"
(define resultCats (solve (assert (interpret* F iCats))))
; Lift the model back to atoms in Un
(interpretation->relations (evaluate iCats resultCats)) ; => cats: b
```

The details are unimportant but at a high-level the code: defines a relation `cats`; compiles the relation to symbolic values via `instantiate-bounds`; uses `solve` to find a concrete satisfying assignment to the symbolic values; uses `evaluate` to replace the symbolic values with the concrete ones; and finally uses `interpretation->relations` to lift the result back to the universe of "cats".

The final call to `interpretation->relations` utilizes unlifted code and thus expects concrete inputs. Calling the function with a symbolic interpretation, however, does not produce an error:

```
(interpretation->relations iCats) ; => cats: a,b,c,d (WRONG)
```

Instead, it silently returns the wrong answer because the symbolic values are mistakenly interpreted as "true" values, in a manner similar to the first example from section 1 of the paper. With our typed version, the erroneous code produces a type error:

```
#lang typed/rosette
(interpretation->relations iCats) ; => TYERR: expected concrete value
```

Other parts of Ocelot exposed a gap in our type system involving Racket `structs`, a kind of named record. A struct definition may "inherit" properties from a base struct and thus requires record subtyping. Our occurrence typing, however, currently does not support refining types with this kind of subtyping and thus we needed casts in a few places to fully type check this library.

6.3 Synthesizing Incremental Algorithms

Incremental algorithms speed up programs by avoiding full recomputation when successive inputs are related. We ported an SDSL for synthesizing incremental algorithms [Shah and Bodik 2017][11] from Rosette to Typed Rosette. The language utilizes unsafe Rosette, in particular hash tables, but uses dynamic checks to prevent symbolic values from reaching unlifted constructs.

Dynamic checks can be unreliable, however, since they are sometimes pruned and thus unreported by Rosette. Further, we discovered (and the lead author acknowledged) that some checks were erroneous or incomplete, meaning that symbolic values could still reach unlifted positions. By porting the code to Typed Rosette, we were able to more completely and reliably specify where

[10] https://jamesbornholt.github.io/ocelot/

[11] Thanks to Rohin Shah for providing us access to the code repository.

symbolic values are not allowed. In addition, we could remove approximately two dozen of the manually-inserted checks (approximately 100 lines).

Porting this library required some effort, however, since it utilizes features, like `eval`, unavailable in Typed Rosette. To accommodate these features, we had to leave some code untyped, and instead used a typed "shim" layer as described in section 5.5. In addition, we had to further help the type checker in two ways. The first annotates values with symbolic types for mutation purposes, e.g.:

```
#lang typed/rosette
(define v (make-vector 7 (ann #f : (U Bool))))
(define-symbolic* b : Bool)
(if b (vector-set! v 0 #t) (vector-set! v 1 #t))
```

This code creates a 7-element vector initialized with concrete false values and would thus be assigned a concrete type. We wanted to mutate the vector in a symbolic path, however, so we needed to annotate the initialization value with a symbolic type.

The second class of annotations we needed involved hash tables. Specifically, the result of a hash lookup has a union type because the lookup may return false if the lookup fails. Thus to use the result of the lookup, we need an extra occurrence typing check to eliminate the false case:

```
#lang typed/rosette
(define h (hash 'a 0 'b 1))
(let ([x (hash-ref h 'a #f)]) ; x has type (U Int False) due to fail case
  (if (false? x) (error "failed!") (+ x 1))) ; x has type Int
```

6.4 Vector Programming (A Problematic Example)

SynthCL is a Rosette-hosted SDSL that helps programmers verify and synthesize OpenCL programs. It represents a less ideal, yet interesting test case for Typed Rosette. Specifically, the language has a C-like type system that is largely incompatible with Typed Rosette's types. While we were able to port the language to Typed Rosette, we had to reimplement large parts of its the existing type system. In the future, we hope to explore the *extensibility* of Typed Rosette, and whether it can more smoothly accommodate typed languages such as this one.

7 RELATED WORK

Handling Unlifted Constructs Most symbolic execution engines prevent invalid interaction of symbolic values with unlifted constructs by either equipping an entire language to handle symbolic values or limiting programmers to a safe subset. For example, Symbolic Pathfinder [Păsăreanu et al. 2008] symbolically executes Java programs with a replacement JVM supporting symbolic values. This approach is the more difficult, however, and may produce complex solver encodings. Alternatively, the Leon [Blanc et al. 2013] and Kaplan [Köksal et al. 2012] languages restrict programmers to a subset of Scala called PureScala. Similarly, Rubicon [Near and Jackson 2012] symbolically executes only a subset of Ruby but does not support interacting with other parts of the language. This approach limits the expressiveness of the language and thus possible applications of the tool.

The above approaches of ensuring safe behavior of symbolic values may still be insufficient if the program interacts with components, like libraries, outside the control of the symbolic execution engine. Concolic testing frameworks in particular have devised various heuristics to handle this situation. The EXE [Cadar et al. 2006] framework, a tool for testing C programs, handles uninstrumented parts of the program by logging calls into uninstrumented functions, instrumenting the code, rerunning the program, and then repeating the process until there are no more uninstrumented function calls. It's not clear, however, if this algorithm accommodates

dynamically-linked libraries. Other tools, like CUTE [Sen et al. 2005] and KLEE [Cadar et al. 2008] concretize symbolic values when they flow to library code. This approach does not consider all program paths and thus may not work for some applications of symbolic execution such as verification. The DART [Godefroid et al. 2005] tool concretizes symbolic values but trades incompleteness for possible non-termination. Specifically, when encountering uninstrumented code, execution falls back to the concrete result and the tool sets an "incompleteness" flag to true. Testing continues until all branches are covered; otherwise the tool runs forever.

SMTen resembles Rosette in that it allows specifying search problems with a high-level functional language, Haskell, which is then compiled to solver encodings. Programmers do not explicitly compute with symbolic values, however, making SMTen less general than Rosette. Thus, although it is a typed language, SMTen does not utilize symbolic types in the manner of Typed Rosette.

Information Flow Our path markers—concrete or symbolic—resemble the "pc" label [Denning 1982] used by information flow type systems [Myers 1999; Pottier and Simonet 2003] to track implicit flows [Denning and Denning 1977]. The two systems differ, however, because label checking and type checking are roughly independent in information flow analyses. In other words, labels are arbitrary tags added *on top of* values and any value may have any label, e.g., "high" or "low", depending on its program context. Thus, the "pc" may be computed from only the program flow and labels on values, but does not need type information. In contrast, our "labels" are *intrinsic* to values, e.g., a symbolic value introduced with `define-symbolic` may never have a concrete type. Thus type and label checking are more intertwined in our type system; for example in conditional branches, the path is marked symbolic only if the test expression is symbolic *and* has boolean type. Finally, our concreteness polymorphism allows label-based overloading; this feature appears unavailable even in more practical information flow-checking languages like JFlow [Myers 1999].

8 CONCLUSION AND FUTURE WORK

We advocate for "lenient symbolic execution", which equips only a small language subset to handle symbolic values, yet exposes a full range of features to programmers. Such a language is easier to implement and produces simpler solver encodings, yet does not limit programmers to an impractically restrictive language. To help write correct programs in such a mixed system, a type system reports when symbolic values unexpectedly reach unlifted positions. Specifically, we developed $\widehat{\lambda}_{\bullet}$, a typed λ-calculus whose types distinguish symbolic from concrete values, and we created Typed Rosette, a typed version of the solver-aided Rosette language, which supports lenient symbolic execution. Our evaluation of Typed Rosette via a comprehensive test suite demonstrates that our calculus is precise and useful enough to model a practical language. In the future, we hope to explore whether the type information available in Typed Rosette can be of further use when implementing specific solver-aided tasks, for example type-directed program synthesis.

ACKNOWLEDGMENTS

This work is supported by the National Science Foundation under Grant Nos. 1518844 and 1651225. We thank James Bornholt, Ras Bodik, and the UWPLSE group for insightful conversations about Rosette; Matthias Felleisen for reading drafts, Ben Greenman for advice on notation, and the POPL reviewers for their attention to detail in helping to polish the paper into its final form.

REFERENCES

Régis Blanc, Viktor Kuncak, Etienne Kneuss, and Philippe Suter. 2013. An Overview of the Leon Verification System: Verification by Translation to Recursive Functions. In *Proceedings of the 4th Workshop on Scala*. 1:1–1:10.

James Bornholt and Emina Torlak. 2017. Synthesizing Memory Models from Framework Sketches and Litmus Tests. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI 2017).*

467–481.

James Bornholt, Emina Torlak, Dan Grossman, and Luis Ceze. 2016. Optimizing Synthesis with Metasketches. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 775–788.

Robert S. Boyer, Bernard Elspas, and Karl N. Levitt. 1975. SELECT—a Formal System for Testing and Debugging Programs by Symbolic Execution. In *Proceedings of the International Conference on Reliable Software*. 234–245.

Cristian Cadar, Daniel Dunbar, and Dawson Engler. 2008. KLEE: Unassisted and Automatic Generation of High-coverage Tests for Complex Systems Programs. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. 209–224.

Cristian Cadar, Vijay Ganesh, Peter M. Pawlowski, David L. Dill, and Dawson R. Engler. 2006. EXE: Automatically Generating Inputs of Death. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*. 322–335.

Stephen Chang, Alex Knauth, and Ben Greenman. 2017. Type Systems As Macros. In *Proceedings of the 44th ACM SIGPLAN Symposium on Principles of Programming Languages*. 694–705.

Dorothy E. Denning and Peter J. Denning. 1977. Certification of Programs for Secure Information Flow. *Commun. ACM* 20, 7 (1977), 504–513.

Dorothy Elizabeth Robling Denning. 1982. *Cryptography and Data Security*. Addison-Wesley Longman Publishing Co., Inc.

Greg Dennis, Felix Sheng-Ho Chang, and Daniel Jackson. 2006. Modular Verification of Code with SAT. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis*. 109–120.

Azadeh Farzan, Andreas Holzer, Niloofar Razavi, and Helmut Veith. 2013. Con2Colic Testing. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 37–47.

Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. 2009. *Semantics Engineering with PLT Redex*. MIT Press.

Matthew Flatt. 2002. Composable and Compilable Macros: You Want It when?. In *Proceedings of the Seventh ACM SIGPLAN International Conference on Functional Programming*. 72–83.

Matthew Flatt. 2016. Binding As Sets of Scopes. In *Proceedings of the 43rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 705–717.

Matthew Flatt and PLT. 2010. *Reference: Racket*. Technical Report PLT-TR-2010-1. PLT Design Inc. https://racket-lang.org/tr1/.

Tim Freeman and Frank Pfenning. 1991. Refinement Types for ML. In *Proceedings of the ACM SIGPLAN 1991 Conference on Programming Language Design and Implementation*. 268–277.

Patrice Godefroid, Nils Klarlund, and Koushik Sen. 2005. DART: Directed Automated Random Testing. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 213–223.

Patrice Godefroid, Michael Y. Levin, and David Molnar. 2012. SAGE: Whitebox Fuzzing for Security Testing. *Queue* 10, 1 (2012), 20:20–20:27.

Sumit Gulwani, Susmit Jha, Ashish Tiwari, and Ramarathnam Venkatesan. 2011. Synthesis of Loop-free Programs. In *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 62–73.

Daniel Jackson. 2002. Alloy: A Lightweight Object Modelling Notation. *ACM Trans. Softw. Eng. Methodol.* 11, 2 (2002), 256–290.

Joxan Jaffar, Vijayaraghavan Murali, Jorge A. Navas, and Andrew E. Santosa. 2012. TRACER: A Symbolic Execution Tool for Verification. In *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*. 758–766.

James C. King. 1975. A New Approach to Program Testing. In *Proceedings of the International Conference on Reliable Software*. 228–233.

Ali Sinan Köksal, Viktor Kuncak, and Philippe Suter. 2012. Constraints As Control. In *Proceedings of the 39th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 151–164.

Andrew C. Myers. 1999. JFlow: Practical Mostly-static Information Flow Control. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 228–241.

Joseph P. Near and Daniel Jackson. 2012. Rubicon: Bounded Verification of Web Applications. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. 60:1–60:11.

Stuart Pernsteiner, Calvin Loncaric, Emina Torlak, Zachary Tatlock, Xi Wang, Michael D. Ernst, and Jonathan Jacky. 2016. Investigating Safety of a Radiotherapy Machine Using System Models with Pluggable Checkers. In *Computer Aided Verification: 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*. 23–41.

Phitchaya Mangpo Phothilimthana, Tikhon Jelvis, Rohin Shah, Nishant Totla, Sarah Chasins, and Rastislav Bodik. 2014. Chlorophyll: Synthesis-aided Compiler for Low-power Spatial Architectures. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 396–407.

Benjamin C. Pierce and David N. Turner. 1998. Local Type Inference. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*. 252–265.

François Pottier and Vincent Simonet. 2003. Information Flow Inference for ML. *ACM Trans. Program. Lang. Syst.* 25, 1 (2003), 117–158.

Corina S. Păsăreanu, Peter C. Mehlitz, David H. Bushnell, Karen Gundy-Burlet, Michael Lowry, Suzette Person, and Mark Pape. 2008. Combining Unit-level Symbolic Execution and System-level Concrete Execution for Testing NASA Software. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ACM, New York, NY, USA, 15–26.

Koushik Sen, Darko Marinov, and Gul Agha. 2005. CUTE: A Concolic Unit Testing Engine for C. In *Proceedings of the 10th European Software Engineering Conference Held Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 263–272.

Rohin Shah and Rastislav Bodik. 2017. Automated Incrementalization through Synthesis. In *Proceedings of the First Workshop on Incremental Computing*.

Armando Solar-Lezama, Rodric Rabbah, Rastislav Bodík, and Kemal Ebcioğlu. 2005. Programming by Sketching for Bit-streaming Programs. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*. 281–294.

T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. 2009. Practical Variable-Arity Polymorphism. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held As Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*. 32–46.

Sam Tobin-Hochstadt and Matthias Felleisen. 2010. Logical Types for Untyped Languages. In *Proceedings of the 15th ACM SIGPLAN International Conference on Functional Programming*. 117–128.

Sam Tobin-Hochstadt, Vincent St-Amour, Ryan Culpepper, Matthew Flatt, and Matthias Felleisen. 2011. Languages As Libraries. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation*. 132–141.

Emina Torlak and Rastislav Bodik. 2013. Growing Solver-aided Languages with Rosette. In *Proceedings of the 2013 ACM International Symposium on New Ideas, New Paradigms, and Reflections on Programming & Software*. 135–152.

Emina Torlak and Rastislav Bodik. 2014. A Lightweight Symbolic Virtual Machine for Solver-Aided Host Languages. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 530–541.

Konstantin Weitz, Doug Woos, Emina Torlak, Michael D. Ernst, Arvind Krishnamurthy, and Zachary Tatlock. 2016. Scalable Verification of Border Gateway Protocol Configurations with an SMT Solver. In *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*. 765–780.

A.K. Wright and M. Felleisen. 1994. A Syntactic Approach to Type Soundness. *Inf. Comput.* 115, 1 (1994), 38–94.