Hybrid Types, Invariants, and Refinements For Imperative Objects

Cormac Flanagan
Department of Computer Science
University of California, Santa Cruz

Stephen N. Freund
Department of Computer Science
Williams College

Aaron Tomb
Department of Computer Science
University of California, Santa Cruz

Abstract

To control the complexity of large object-oriented systems, objects should communicate via precisely-specified interfaces. Static type checking catches many interface violations early in the development cycle, but decidability limitations preclude checking all desired properties statically. In contrast, dynamic checking supports expressive specifications but may miss errors on execution paths that are not tested. We present a hybrid approach for checking precise object specifications that reasons statically, where possible, but also dynamically, when necessary. This hybrid approach supports a rich specification language with features such as object invariants and refinement types.

1. Introduction

The construction and validation of large software systems is extremely challenging. To control the complexity of such systems, they are ideally constructed as a collection of objects that cooperate via precisely-specified interfaces. These interface specifications then enable individual objects to change and evolve without the need to understand the entire system. Thus, a key challenge in the construction of any object-oriented system is to precisely specify object interfaces, and to verify (to the degree possible) that object implementations respect these interfaces.

Types can naturally describe some aspects of object interfaces. An advantage of these type specifications is that static type checking can then catch many errors at compile time, when they are cheaper to fix. However, static type checking is possible only for certain kinds of specifications, and extending the type language to support expressive specifications, such as arbitrary method preconditions and postconditions, results in a type system that is not statically decidable. Hence, traditional object type systems cannot express many useful interface specifications, such as:

- The method `getCount` returns a positive number.
- The field `i` of an object is a valid index into the array in the field `a`.
- This object contains a sorted list.
- The method `draw` expects a `Point` object that is within a specified rectangle.

Other static checkers suffer from similar limitations. For example, ESC/Java [14] may reject valid programs due to the incompleteness of its theorem prover.

Presented at the FOOL/WOOD Workshop, January, 2006.

In contrast, precise object specifications are straightforward to check dynamically, either via assert statements [37] or via specialized contract languages [29, 11, 24, 16, 20, 26, 34, 21]. However, dynamic checking imposes a certain performance overhead. More seriously, dynamic checking only detects errors on code paths and data values of actual executions. Thus, dynamic checking may not catch errors until very late in the development cycle, or possibly post-deployment, when they are significantly more expensive to fix.

In summary, decidable static type checkers provide complete checking of limited specifications, while dynamic approaches provide limited coverage of expressive specifications. In this paper, we explore a *hybrid* approach to checking expressive object specifications that combines the advantages of these prior purely-static and purely-dynamic approaches. In a nutshell, hybrid type checking involves:

1. embracing expressive type languages that can document precise object interfaces, but which are therefore statically undecidable;
2. leveraging static type checking to detect as many errors as possible at compile time; and
3. leveraging dynamic checking to check any remaining (not statically verified) correctness properties at run time.

Our specification language extends an Abadi-Cardelli-style type system [1] with *refinement types* and *object invariants.* A refinement type defines a subset or refinement of an existing type [28, 15, 8, 43, 42]. For example, the refinement type $\{x : \texttt{Int} \,|\, x > 0\}$ denotes the set of positive integers. For expressiveness, our type system requires only that refinement predicates be *pure*; that is, refinement predicates are arbitrary program expressions that do not access mutable data. Each object type also has an associated predicate, called an *object invariant*, which must hold on all objects of that type.

We formalize the semantics of this expressive specification language using a type system that is sound but necessarily undecidable. To enable useful checking, we then present a hybrid type checking algorithm with the following desirable features:

1. The hybrid type checker statically rejects as many ill-typed programs as possible.
2. Due to decidability limitations, the hybrid type checker may statically accept some *subtly* ill-typed programs, but it will insert sufficient dynamic casts to guarantee that their specifications are never violated. Any attempted specification violation is thus either caught statically

(where possible) or via dynamic checks (where necessary).

3. The output of the hybrid type checker is always a well-typed program (and so, for example, type-directed optimizations are applicable).

In previous work, we explored this hybrid strategy in the simpler context of the pure functional lambda-calculus [12, 18]. To help verify large object-oriented systems, this paper extends these earlier ideas to a more interesting imperative language that supports objects (with both final and mutable methods, and imperative method update) and expressive object specifications (with object subtyping, refinement types, and object invariants).

This initial work studies hybrid type checking in the context of an idealized object language, but we hope that this work will serve as a foundation for exploring more complex object languages and type systems. Our long term goal is to apply hybrid type checking to realistic object-oriented programming languages, such as Java [17], C# [10], or Objective CAML [35].

The presentation of our results proceeds as follows. The next section informally describes our object language and type system. Section 3 applies this type language to document some precise specifications. Section 4 and 5 formally describe the operational semantics and (undecidable) type system for this language. Section 6 presents our hybrid type checking algorithm, and Section 7 states the key correctness properties of our type system and compilation algorithm. Section 8 discusses related work, and Section 9 concludes.

2. The Hoop Language

We present our approach to hybrid checking of expressive object specifications in terms of Hoop, an idealized hybrid object-oriented programming language based on the calculi of Abadi and Cardelli [1]. The syntax of this language is shown in Figure 1.

2.1 Types

We begin with an overview of the type language, since it contains a number of features that are unusual in object type systems, such as dependent types, refinement types, and object invariants.

A base type B is either `Bool`, `Int`, or `Unit`. Since these base types are fairly coarse and cannot, for example, express integer subranges, we introduce *refinement types* such as $\{x : \texttt{Int} \mid x > 0\}$, which denotes the set of positive integers. More generally, the refinement type $\{x : B \mid t\}$ denotes the set of values c of type B that satisfy the boolean predicate t, *i.e.*, for which the term $t[x := c]$ evaluates to true. We use a base type B as an abbreviation for the refinement type $\{x : B \mid \texttt{true}\}$.

Our refinement types are inspired by prior work on decidable refinement type systems [28, 15, 8, 43, 42] but are more expressive, which causes type checking to be undecidable. In particular, subtyping between two refinement types $\{x : B \mid t_1\}$ and $\{x : B \mid t_2\}$ reduces to checking implication between the corresponding (arbitrary) refinement predicates, which is clearly undecidable. These decidability difficulties are circumvented by our hybrid type checking algorithm.

An object type in our language has the form

$$x : [l_i : f_i \cdot p_i \cdot (y : T_i \to T_i')\ ^{i \in 1..n}].t$$

This type denotes an object containing n methods with distinct names $l_{1..n}$. (As usual, fields can be encoded as

Figure 1: Syntax

$s, t, u ::=$	*Terms:*
x	variable
d	object
v	value
$t.l(s)$	select
$t.l \Leftarrow \varsigma(x, y)u$	update
$\langle T \rangle\ t$	cast
$\texttt{let}\ x = s\ \texttt{in}\ t\ \texttt{as}\ T$	binding
$d ::=$	*Objects:*
$x.[l_i(y) = t_i\ ^{i \in 1..n}]\ \texttt{as}\ T$	object
$v, w ::=$	*Values:*
c	constant
a	object address
$a\ \texttt{view}\ T$	object view
$S, T, U ::=$	*Types:*
$\{x : B \mid t\}$	base refinement type
$x : [l_i : f_i \cdot p_i \cdot M_i\ ^{i \in 1..n}].t$	object type
$M, N ::=$	*Method Types:*
$y : T_i \to T_i'$	method type
$B ::= \texttt{Unit} \mid \texttt{Bool} \mid \texttt{Int}$	*Base types:*
$f ::= \texttt{final} \mid \texttt{mutable}$	*Final modifiers:*
$p ::= \texttt{pure} \mid \texttt{impure}$	*Purity modifiers:*

methods.) Each method l_i has the dependent type $(y : T_i \to T_i')$, meaning that it takes an argument of type T_i, and returns a result of type T_i', where the formal parameter y may occur free in T_i'. We abbreviate $(y : T_i \to T_i')$ by $(T_i \to T_i')$ if y does not occur free in T_i'. Since our type system contains dependent types that may refer to variables, each object type includes a binding for the self-reference variable x (often called `this`) that may be mentioned in method types. Each method l_i also has two method qualifiers f_i and p_i.

- The method qualifier $f_i \in \{\texttt{final}, \texttt{mutable}\}$ indicates whether the corresponding method can be updated. Object subtyping is invariant on `mutable` methods, and covariant on immutable (or `final`) methods.
- The method qualifier $p_i \in \{\texttt{pure}, \texttt{impure}\}$ indicates whether the method l_i is *pure*. Specifically, an expression is pure if its evaluation never accesses (*i.e.*, invokes or updates) a `mutable` method. Purity qualifiers are ordered by $\texttt{pure} \sqsubset \texttt{impure}$, since it is safe to consider a pure method to be impure.

Each object type includes an *object invariant* t, which is a boolean predicate over the self-reference variable x. This object invariant is guaranteed to hold for any object of this type, and so cannot evaluate to `false`. To ensure that this guarantee holds in the presence of method updates, we require that every object invariant be pure. Similarly, refinement predicates must also be pure.

2.2 Terms

Hoop source terms include variables, constants, objects, method invocation and update, casts, and let-expressions. An object d has the form

$$x.[l_i(y) = t_i \ ^{i\in 1..n}] \text{ as } T$$

and consists of a collection of methods $l_{1..n}$ with corresponding method bodies $t_{1..n}$, where x and y provide bindings for the self-reference variable and the formal parameter, respectively. Each object is annotated with an explicit type T.

Constants in Hoop include boolean and integer constants, as well as operations such as $+$, $\geq$, **and**, and **not**. Although our examples use infix notation for primitive operations, Hoop internally represents these primitive operations as method invocations. A cast $\langle T\rangle\ t$ dynamically converts (where possible) the value produced by t to type T, or else fails. For technical reasons, `let` bindings include an explicit type annotation.

Hoop also includes object addresses a and *object views* (a `view` T). These constructs are used to formalize the operational semantics of the language, and should not appear in source programs.

3. Examples

In this section, we illustrate our type language and hybrid type checking through several informal examples. For readability, we omit `final` and `pure` modifiers and assume methods are immutable and pure unless specified otherwise.

3.1 Refinement Types

We first introduce some useful refinements of `Int`:

$$\begin{aligned} \texttt{Pos} &\triangleq \{z\!:\!\texttt{Int} \mid z > 0\} \\ \texttt{Nat} &\triangleq \{z\!:\!\texttt{Int} \mid z \geq 0\} \end{aligned}$$

Since $(z > 0) \Rightarrow (z \geq 0)$, the expected subtype relation $\texttt{Pos} <: \texttt{Nat}$ holds. Similarly, since $(z \geq 0) \Rightarrow \texttt{true}$, we also have that $\texttt{Nat} <: \texttt{Int}$.

Subtyping between refinement types is undecidable in general, and therefore so is type checking. For example, consider the method update:

$$p.m \Leftarrow \varsigma(x, y)t$$

where x and y provide bindings for the self-reference variable and formal parameter, respectively. Suppose the mutable method $p.m$ has type $\texttt{Int} \rightarrow \texttt{Nat}$, and that the term t has type $\{z : \texttt{Int} \mid z = y^2\}$. To check if this method update is well-typed, the compiler needs to decide the subtype relation

$$\{z\!:\!\texttt{Int} \mid z = y^2\} <: \texttt{Nat}$$

which in turn reduces to deciding the validity of the implication:

$$(z = y^2) \ \Rightarrow\ (z \geq 0)$$

If the compiler can prove this implication, then t has type `Nat` (via subsumption), and so the method update is well-typed. Conversely, if the compiler can refute this implication, then t does not have type `Nat`, and the compiler can reject the program as ill-typed. (In this case, however, the implication is valid and so cannot be refuted.)

Due to the expressiveness of the Hoop type language, the compiler may encounter situations where its algorithms can neither prove nor refute such implications, and so the compiler cannot decide whether or not expressions are well-typed. A key design question is how the compiler should handle such situations. Optimistically *accepting* such programs means that type specifications cannot be trusted, since they may be violated at run time, which is clearly undesirable. Pessimistically *rejecting* such programs would cause the compiler to reject some well-typed programs, which seems too brittle for use in practice, since it would be difficult to predict which programs the compiler would accept.

Hybrid type checking handles such situations by provisionally accepting the program but inserting a cast that dynamically enforces the specification of $p.m$, yielding the compiled code:

$$p.m \Leftarrow \varsigma(x, y)(\langle \texttt{Nat}\rangle\ t)$$

This compiled code is well-typed, and it dynamically ensures that the specification of $p.m$ is never violated, *i.e.*, the method $p.m$ only returns natural numbers. This example illustrates how hybrid type checking enforces precise specifications, such as $\texttt{Int} \rightarrow \texttt{Nat}$, even if those specifications cannot be always verified statically. In comparison, a static type checker may be able to enforce only weaker specifications, such as $\texttt{Int} \rightarrow \texttt{Int}$.

3.2 Pairs

We next define an object `pair` containing two integers and a method that computes their sum. We assign `pair` a conventional type `Pair`:

$$\begin{aligned} \texttt{pair} &\triangleq p.[a = 3, b = 5, \mathit{sum}(\_) = p.a + p.b] \text{ as } \texttt{Pair} \\ \texttt{Pair} &\triangleq p\!:\![a : \texttt{Int}, b : \texttt{Int}, \mathit{sum} : \texttt{Unit} \rightarrow \texttt{Int}].\texttt{true} \end{aligned}$$

Although Hoop does not directly support fields, we may encode a field of type T as a method of type $\texttt{Unit} \rightarrow T$. We abbreviate such a field type to simply T for readability. For *sum*, we use the parameter name _ to indicate that this value is not used in the method body.

Alternatively, we could use refinement types to specify that the second component of the pair is not less than the first:

$$\texttt{OrderedPair} \ \triangleq\ p\!:\!\begin{bmatrix} a : \texttt{Int}, \\ b : \{z\!:\!\texttt{Int} \mid z \geq p.a\}, \\ \mathit{sum} : \texttt{Unit} \rightarrow \texttt{Int} \end{bmatrix}.\texttt{true}$$

This type specifies that if the method b terminates, it must return an integer greater than $p.a$. Note that the predicate $z \geq p.a$ must be *pure*. Our type system ensures that pure expressions cannot access mutable data. Without this restriction, a program could use subtyping to hide the field b and then break the predicate on b by modifying a. We permit covariant subtyping of immutable fields, meaning that

$$\texttt{OrderedPair} <: \texttt{Pair}$$

An alternative definition of `OrderedPair` uses an object invariant to express the ordering relation between a and b:

$$\texttt{OrderedPair}' \ \triangleq\ p\!:\!\begin{bmatrix} a : \texttt{Int}, \\ b : \texttt{Int}, \\ \mathit{sum} : \texttt{Unit} \rightarrow \texttt{Int} \end{bmatrix}.(p.a \leq p.b)$$

For any object p of this type $\texttt{OrderedPair}'$, the object invariant $(p.a \leq p.b)$ must never return `false`; it can only return `true`, or it may diverge.[1] In particular, the method calls $p.a$ and $p.b$ can diverge without violating this object invariant.

[1] Allowing object invariants to diverge avoids the complexity of reasoning about termination in the type system

We require that the invariant of a subtype imply the invariant of its supertype. For example, since the implication $(p.a \leq p.b) \Rightarrow \texttt{true}$ holds, we have that:

$$\texttt{OrderedPair}' <: \texttt{Pair}$$

We check invariant implication in a context in which we know the precise type of the self-reference variable. Thus, since for all p of type `OrderedPair`, the predicate $(p.a \leq p.b)$ never evaluates to `false`, we can conclude that $\texttt{true} \Rightarrow (p.a \leq p.b)$, and hence that:

$$\texttt{OrderedPair} <: \texttt{OrderedPair}'$$

Thus, subtyping allows us to move information from method specifications to the object invariant.

One interesting difference between object invariants and precise method specifications is that the latter can specify the behavior of mutable methods. For example, the following type allows the method b to be updated with any integer that is not less than a:

$$\texttt{MutableOrderedPair} \triangleq p\!:\!\begin{bmatrix} a : \texttt{Int}, \\ b : \texttt{mutable impure } \{z\!:\!\texttt{Int} \mid z \geq p.a\}, \\ sum : \texttt{impure Unit} \rightarrow \texttt{Int} \end{bmatrix}\!.\texttt{true}$$

3.3 Geometric Objects

As a final example, consider the following types for points, rectangles, and squares:

$$\begin{aligned} \texttt{Point} &\triangleq p\!:\![x : \texttt{Int}, y : \texttt{Int}].\texttt{true} \\ \texttt{Rectangle} &\triangleq r\!:\![b : \texttt{Point}, w : \texttt{Nat}, h : \texttt{Nat}].\texttt{true} \\ \texttt{Square} &\triangleq r\!:\![b : \texttt{Point}, w : \texttt{Nat}, h : \texttt{Nat}].(r.w = r.h) \end{aligned}$$

Clearly, $\texttt{Square} <: \texttt{Rectangle}$. The following subtype of `Point` allows us to describe points must lie within a given `Rectangle` r:

$$\texttt{PtInRect}(r) \triangleq p\!:\![x : \texttt{Int}, y : \texttt{Int}].\left(\begin{array}{cc} & (r.b.x) \leq p.x \leq (r.b.x + r.w) \\ \wedge & (r.b.y) \leq p.y \leq (r.b.y + r.h) \end{array}\right)$$

We can also specify a type of mutable points, and a subtype of mutable points that must stay within a given rectangle:

$$\texttt{MutablePoint} \triangleq p\!:\!\begin{bmatrix} x : \texttt{mutable impure Int}, \\ y : \texttt{mutable impure Int} \end{bmatrix}\!.\texttt{true}$$

$$\texttt{MutablePtInRect}(r) \triangleq p\!:\!\begin{bmatrix} x : \texttt{mut. imp. } \{z\!:\!\texttt{Int} \mid r.b.x \leq z \leq r.b.x + r.w\}, \\ y : \texttt{mut. imp. } \{z\!:\!\texttt{Int} \mid r.b.y \leq z \leq r.b.y + r.h\} \end{bmatrix}\!.\texttt{true}$$

The compiler will ensure that all updates to an object of type $\texttt{MutablePtInRect}(r)$ are within the specified bounds, either via static reasoning, where possible, or else via implicit dynamic checks, if necessary.

We can also specify more general relationships, such as the type of all points occurring within a particular `Shape`, where `Shape` has subtypes `Rectangle` and `Circle`, all of which implement a *contains* method:

$$\begin{aligned} \texttt{Shape} &\triangleq p\!:\![contains : \texttt{Point} \rightarrow \texttt{Bool}].\texttt{true} \\ \texttt{Rectangle} &\triangleq p\!:\![contains : \texttt{Point} \rightarrow \texttt{Bool},\ \ldots].\texttt{true} \\ \texttt{Circle} &\triangleq p\!:\![contains : \texttt{Point} \rightarrow \texttt{Bool},\ \ldots].\texttt{true} \end{aligned}$$

The following type then characterizes points that must lie within a particular `Shape`:

$$\texttt{PtInShape}(s) \triangleq p\!:\![x : \texttt{Int}, y : \texttt{Int}].(s.contains(p))$$

4. Hoop Operational Semantics

We formalize the run-time behavior of Hoop programs using the small-step operational semantics shown in Figure 2. Evaluation is performed inside evaluation contexts, which are defined by the grammar:

$$\begin{aligned} \mathcal{E} \ ::= \ & \bullet \mid \mathcal{E}.l(t) \mid v.l(\mathcal{E}) \mid \mathcal{E}.l \Leftarrow \varsigma(x,y)u \\ & \mid \langle T \rangle\ \mathcal{E} \mid \texttt{let } x = \mathcal{E} \texttt{ in } t \texttt{ as } T \end{aligned}$$

A *store* σ maps object addresses a to objects d, and we use $\emptyset$ to denote the empty store. A state (σ, t) is a pair of a store and a term. The relation $(\sigma, t) \longrightarrow (\sigma', t')$ performs a single evaluation step, and the relation $\longrightarrow^*$ is the transitive closure of $\longrightarrow$.

The first five evaluation rules are fairly straightforward. The rule [E-Obj] adds a new object to the store at a fresh address. The rule [E-Sel-Obj] for a method invocation $a.l_j(v)$ extracts the method body t_j from the object at address a, and replaces the self-reference variable x and the formal parameter y with the object address a and the argument v, respectively. The rule [E-Upd] for a method update $a.l_j \Leftarrow \varsigma(x,y)u$ replaces the method l_j in the object at address a. We use α-renaming to implicitly match the self-reference and parameter variables of the original and new methods. The rule [E-Let] for $(\texttt{let } x = v \texttt{ in } t \texttt{ as } T)$ simply replaces x by v in t.

Constants include both *basic constants*, such as `true` and 3, and *primitive operations*, such as `not` and `+`. Primitive operations are actually objects with an *apply* method that provides the required functionality. The prefix and infix syntax for primitive operations shown in the earlier examples is actually desugared into invocations of *apply* methods, as follows:

$$\begin{aligned} \texttt{not true} &\triangleq \texttt{not}.apply(\texttt{true}) \\ 3 + 4 &\triangleq +.apply(\ (3 : \texttt{Int}, 4 : \texttt{Int})\ ) \end{aligned}$$

The desugaring for $+$ uses the following notations for creating objects representing pair of values:

$$\begin{aligned} S * T &\triangleq x\!:\![fst : S, snd : T].\texttt{true} \\ (v_1 : S, v_2 : T) &\triangleq x.[fst = v_1, snd = v_2] \text{ as } (S * T) \end{aligned}$$

The rule [E-Const] evaluates a method invocation $c.l(v)$ on a primitive operation c, and relies on the meaning function $[\![\cdot]\!]$ to define the behavior of primitive operations. Specifically, $[\![c]\!](l, v, \sigma)$ returns the result of invoking the method l of the primitive operation c on the argument v in store σ. For example, if $\sigma(a)$ contains the pair $(3 : \texttt{Int}, 4 : \texttt{Int})$, we have:

$$\begin{aligned} [\![\texttt{not}]\!](apply, \texttt{true}, \sigma) &= \texttt{false} \\ [\![+]\!](apply, a, \sigma) &= 7 \\ [\![\leq]\!](apply, a, \sigma) &= \texttt{true} \\ [\![\texttt{and}]\!](apply, a, \sigma) &= \textit{undefined} \end{aligned}$$

The rule [E-CastBase] casts a constant c to refinement type $\{x\!:\!B \mid t\}$ by checking that the predicate t holds on c, *i.e.*, that $t[x := c]$ evaluates to `true`. If $t[x := c]$ returns `false` or diverges, then the cast is said to *fail*.

Casts on object types are more complicated, since we need to check that every method of the resulting casted object returns a value of the appropriate type for all possible argument values. This check is performed in a lazy manner

Figure 2: Evaluation Rules

Evaluation $\boxed{(\sigma, t) \longrightarrow (\sigma', t')}$

In the following rules, we assume:

$$\begin{array}{rcl} d & = & x.[l_i(y) = t_i \ ^{i \in 1..n}] \text{ as } S \\ S & = & x:[l_i : f_i \cdot q_i \cdot (y:S_i \to S_i') \ ^{i \in 1..n}].s' \\ T & = & x:[l_i : f_i \cdot p_i \cdot (y:T_i \to T_i') \ ^{i \in 1..m}].t' \\ T' & = & x:[l_i : f_i \cdot p_i \cdot (y:T_i \to T_i') \ ^{i \in 1..m}].\mathtt{true} \end{array}$$

$$\frac{a \notin \text{dom}(\sigma)}{(\sigma, \mathcal{E}[d]) \longrightarrow (\sigma[a := d], \mathcal{E}[a])} \quad \text{[E-OBJ]}$$

$$\frac{\sigma(a) = d}{(\sigma, \mathcal{E}[a.l_j(v)]) \longrightarrow (\sigma, \mathcal{E}[t_j[x := a, y := v]])} \quad \text{[E-SEL-OBJ]}$$

$$\frac{\begin{array}{c} \sigma(a) = d \\ \begin{array}{rl} d' = & x.[l_i(y) = t_i \ ^{i \in 1..j-1}, \\ & l_j(y) = u, \\ & l_i(y) = t_i \ ^{i \in j+1..n}] \text{ as } S \end{array} \end{array}}{\begin{array}{c}(\sigma, \mathcal{E}[a.l_j \Leftarrow \varsigma(x,y)u]) \\ \longrightarrow (\sigma[a := d'], \mathcal{E}[\mathtt{unit}])\end{array}} \quad \text{[E-UPD-OBJ]}$$

$$\frac{}{\begin{array}{c}(\sigma, \mathcal{E}[\mathtt{let}\ x = v\ \mathtt{in}\ t\ \mathtt{as}\ T]) \\ \longrightarrow (\sigma, \mathcal{E}[t[x := v]])\end{array}} \quad \text{[E-LET]}$$

$$\frac{[\![c]\!](l, v, \sigma) = v'}{(\sigma, \mathcal{E}[c.l(v)]) \longrightarrow (\sigma, \mathcal{E}[v'])} \quad \text{[E-CONST]}$$

$$\frac{(\sigma, t[x := c]) \longrightarrow^* (\sigma', \mathtt{true})}{(\sigma, \mathcal{E}[\langle\{x:B \mid t\}\rangle\ c]) \longrightarrow (\sigma, \mathcal{E}[c])} \quad \text{[E-CAST-BASE]}$$

$$\frac{\begin{array}{c}\sigma(a) = d \qquad m \le n \qquad q_i \sqsubseteq p_i \quad \forall i \in 1..m \\ (\sigma, t'[x := (a\ \mathtt{view}\ T')]) \longrightarrow^* (\sigma', \mathtt{true})\end{array}}{(\sigma, \mathcal{E}[\langle T\rangle\ a]) \longrightarrow (\sigma, \mathcal{E}[a\ \mathtt{view}\ T])} \quad \text{[E-CAST-OBJ]}$$

$$\frac{\begin{array}{c}\sigma(a) = d \\ v' = \langle S_j[x := a]\rangle\ v \\ R = T_j'[x := (a\ \mathtt{view}\ T), y := v]\end{array}}{\begin{array}{c}(\sigma, \mathcal{E}[(a\ \mathtt{view}\ T).l_j(v)]) \\ \longrightarrow (\sigma, \mathcal{E}[\langle R\rangle\ (a.l_j(v'))])\end{array}} \quad \text{[E-SEL-VIEW]}$$

$$\frac{\begin{array}{c}\sigma(a) = d \\ u' = (\langle S_j'\rangle\ (u[y := \langle T_j\rangle\ y][x := (a\ \mathtt{view}\ T)]))\end{array}}{\begin{array}{c}(\sigma, \mathcal{E}[(a\ \mathtt{view}\ T).l_j \Leftarrow \varsigma(x,y)u]) \\ \longrightarrow (\sigma, \mathcal{E}[a.l_j \Leftarrow \varsigma(x,y)u'])\end{array}} \quad \text{[E-UPD-VIEW]}$$

$$\frac{}{(\sigma, \mathcal{E}[\langle T\rangle\ (a\ \mathtt{view}\ S)]) \longrightarrow (\sigma, \mathcal{E}[\langle T\rangle\ a])} \quad \text{[E-CAST-VIEW]}$$

via *object views*. An object view "(a `view` T)" is a wrapper that ensures that the object $\sigma(a)$ behaves according to the specification of type T. Views are introduced by the rule [E-CAST-OBJ], which evaluates the cast $\langle T\rangle\ a$ to the view (a `view` T). In addition, this rule checks that the object $\sigma(a)$ has all the methods mentioned in T (with compatible modifiers), and that the object invariant of T holds on $\sigma(a)$.

We define special evaluation rules for method invocation and update on views. These rules lazily enforce typing restrictions on method parameters and results by performing appropriate checks on method invocations and updates. The rule [E-SEL-VIEW] for a view invocation $(a\ \mathtt{view}\ T).l_j(v)$ retrieves the type S of a, which may be quite different from the view type T:

$$\begin{array}{rcl} S & = & x:[l_i : f_i \cdot q_i \cdot (y:S_i \to S_i') \ ^{i \in 1..n}].s' \\ T & = & x:[l_i : f_i \cdot p_i \cdot (y:T_i \to T_i') \ ^{i \in 1..m}].t' \end{array}$$

The rule reduces the view invocation $(a\ \mathtt{view}\ T).l_j(v)$ to the following method invocation, which contains two additional casts to ensure type safety:

$$\langle T_j'[x := (a\ \mathtt{view}\ T), y := v]\rangle\ (a.l_j(\langle S_j[x := a]\rangle\ v))$$

The cast $\langle S_j[x := a]\rangle\ v$ transforms the argument v of type T_j to a value of the expected argument type S_j, where self-reference variable x is replaced by object's address a. The cast $\langle T_j'[x := (a\ \mathtt{view}\ T), y := v]\rangle\ (\ldots)$ transforms the method result into the expected type T_j', where self-reference variable x and formal parameter y are replaced by original view (a `view` T) and original argument v, respectively.

These two casts ensure that method invocations on object views preserve type safety. To illustrate this guarantee, suppose:

$$\begin{array}{rcl} \sigma(a) & = & x.[\mathit{fact}(y) = \ldots] \text{ as } \mathtt{Fact} \\ \mathtt{Fact} & \triangleq & x:[\mathit{fact} : \mathtt{Nat} \to \mathtt{Nat}].\mathtt{true} \\ \mathtt{BadFact} & \triangleq & x:[\mathit{fact} : \mathtt{Int} \to \mathtt{Nat}].\mathtt{true} \end{array}$$

and consider the term $(\langle\mathtt{BadFact}\rangle\ a).\mathit{fact}(-3)$, which attempts to use the `BadFact` view to circumvent the `Fact` specification of the object $\sigma(a)$. This term evaluates to $\langle\mathtt{Nat}\rangle\ (a.\mathit{fact}(\langle\mathtt{Nat}\rangle\ -3))$, at which point the cast $(\langle\mathtt{Nat}\rangle\ -3)$ fails. Thus, the casts inserted by [E-SEL-VIEW] enforce the specification of the *fact* method, which states that *fact* should be applied only to natural numbers.

The rule [E-UPD-VIEW] reduces a view update

$$(a\ \mathtt{view}\ T).l_j \Leftarrow \varsigma(x,y)u$$

to a normal method update

$$a.l_j \Leftarrow \varsigma(x,y)u'$$

where u' is the following modified version of u with additional casts:

$$u' = (\langle S_j'\rangle\ (u[y := \langle T_j\rangle\ y][x := (a\ \mathtt{view}\ T)]))$$

Here, the formal parameter y in u is replaced with $\langle T_j\rangle\ y$, which is guaranteed to be of the appropriate type T_j, even though y itself may not be. Similarly, the self-reference variable x is replaced with the view (a `view` T), which is of the expected type T, even though a may not be.[2] Finally, the result of u is cast to type S_j', which is the return type expected by (non-view) invocations of $a.l_j$.

The rule [E-CAST-VIEW] evaluates $\langle T\rangle\ (a\ \mathtt{view}\ S)$ to $\langle T\rangle\ a$; thus, the original view is discarded when cast to a different view.

In summary, casts allow the program to claim that a value of one type can be safely considered to be of another type, and the operational semantics performs sufficient run-time checking to detect if this claim is ever violated. This

[2] The alternative of substituting $(\langle T\rangle\ x)$ for x would require redundant re-evaluation of T's object invariant.

ability to perform dynamic casts is crucial for enabling our hybrid type checking algorithm to convert particularly difficult static checks into dynamic checks, when necessary.

5. The Hoop Type System

We next present the (undecidable) type system for the Hoop language as the collection of the type judgments and rules shown in Figures 3 and 4. Type environments are a sequence of variable-type bindings, and we assume that the variables bound in an environment are distinct.

$$E \ ::= \ \emptyset \ | \ E, x : T$$

The judgment $E \vdash t \,:\, T \,\&\, p$ states that the term t has type T and purity p in environment E. This judgment is defined by the following rules:

- The rule [T-Var] states that variable accesses are always pure, since variables are immutable.
- The rule [T-Obj] deals with object creation, and checks that each method has the appropriate result type and purity. In addition, a `mutable` method is considered `impure`, since any call to that method must be `impure`. Newly-created objects must have the trivial invariant `true`. A stronger invariant can later be added via subtyping or casting.
- The rule [T-Let] for $\texttt{let}\ x = t_1\ \texttt{in}\ t_2\ \texttt{as}\ T$ requires that the type T of t_2 must be well-formed in the environment without x, in order to prevent x from escaping its scope.
- The rule [T-Sel] for a method invocation $t.l_j(u)$ checks that the method l_j has type $y\!:\!T_j \to T_j'$, and that the argument u has type $T_j[x := t]$. The type of the method call is then $T_j'[x := t, y := u]$, where the self-reference variable x and the formal parameter y are replaced by the object t and the actual parameter u, respectively.

  Since the terms t and u may appear in the resulting type, these terms must be pure. Note that if one of these terms, say t, is not pure, the method call $t.l_j(u)$ can be refactored into $(\texttt{let}\ x = t\ \texttt{in}\ x.l_j(u)\ \texttt{as}\ R)$, provided the explicit result type R does not mention x.
- The rule [T-Upd] for a method update checks that the updated method is mutable and that the new method body has the appropriate type and purity.
- The rule [T-Const] assigns the type $ty(c)$ to each constant c. Basic constants have precise refinement types, such as:

$$\begin{array}{rcl} \texttt{true} & : & \{b\!:\!\texttt{Bool} \,|\, b\} \\ \texttt{false} & : & \{b\!:\!\texttt{Bool} \,|\, \texttt{not}\ b\} \\ n & : & \{m\!:\!\texttt{Int} \,|\, m = n\} \end{array}$$

  Primitive operations are assigned object types that precisely characterize their behavior. For example, the following type for $+$ states that the method $+.apply$ takes two integer arguments, which are passed in the *fst* and *snd* fields of the pair object y, and that the method result is an integer r that is equal to $(y.\mathit{fst} + y.\mathit{snd})$.

$$\begin{array}{rcl} + & : & x\!:\!\left[\mathit{apply} : \left(\begin{array}{l} y\!:\ (\texttt{Int} * \texttt{Int}) \to \\ \{r\!:\!\texttt{Int} \,|\, r = y.\mathit{fst} + y.\mathit{snd}\} \end{array} \right)\right].\texttt{true} \\ \Leftrightarrow & : & x\!:\!\left[\mathit{apply} : \left(\begin{array}{l} y\!:\ (\texttt{Bool} * \texttt{Bool}) \to \\ \{r\!:\!\texttt{Bool} \,|\, r \Leftrightarrow (y.\mathit{fst} \Leftrightarrow y.\mathit{snd})\} \end{array} \right)\right].\texttt{true} \\ \texttt{not} & : & x\!:\!\left[\mathit{apply} : \left(\begin{array}{l} y\!:\ \texttt{Bool} \to \\ \{r\!:\!\texttt{Bool} \,|\, r \Leftrightarrow \texttt{not}\ y\} \end{array} \right)\right].\texttt{true} \\ = & : & x\!:\!\left[\mathit{apply} : \left(\begin{array}{l} y\!:\ (\texttt{Int} * \texttt{Int}) \to \\ \{r\!:\!\texttt{Bool} \,|\, r \Leftrightarrow (y.\mathit{fst} = y.\mathit{snd})\} \end{array} \right)\right].\texttt{true} \end{array}$$

Figure 3: Type Rules

Type Rules for Terms $\boxed{E \vdash t \,:\, T \,\&\, p}$

$$\frac{(x : T) \in E}{E \vdash x \,:\, T \,\&\, \texttt{pure}} \qquad \text{[T-Var]}$$

$$\frac{\begin{array}{c} E \vdash T \\ T = x\!:\![l_i : f_i \cdot p_i \cdot (y\!:\!T_i \to T_i')\ ^{i \in 1..n}].\texttt{true} \\ f_i = \texttt{mutable} \ \Rightarrow\ p_i = \texttt{impure} \quad \forall i \in 1..n \\ E, x : T, y : T_i \vdash s_i \,:\, T_i' \,\&\, p_i \quad \forall i \in 1..n \end{array}}{E \vdash (x.[l_i(y) = s_i\ ^{i \in 1..n}]\ \texttt{as}\ T) \,:\, T \,\&\, \texttt{pure}} \qquad \text{[T-Obj]}$$

$$\frac{\begin{array}{c} E \vdash t_1 \,:\, S \,\&\, p_1 \\ E, x : S \vdash t_2 \,:\, T \,\&\, p_2 \\ E \vdash T \end{array}}{E \vdash \texttt{let}\ x = t_1\ \texttt{in}\ t_2\ \texttt{as}\ T \,:\, T \,\&\, (p_1 \sqcup p_2)} \qquad \text{[T-Let]}$$

$$\frac{\begin{array}{c} E \vdash t \,:\, T \,\&\, \texttt{pure} \\ T = x\!:\![l_i : f_i \cdot p_i \cdot (y\!:\!T_i \to T_i')\ ^{i \in 1..n}].s \\ E \vdash u \,:\, T_j[x := t] \,\&\, \texttt{pure} \qquad j \in 1..n \end{array}}{E \vdash t.l_j(u) \,:\, T_j'[x := t, y := u] \,\&\, p_j} \qquad \text{[T-Sel]}$$

$$\frac{\begin{array}{c} E \vdash t \,:\, T \,\&\, p \\ T = x\!:\![l_i : f_i \cdot p_i \cdot (y\!:\!T_i \to T_i')\ ^{i \in 1..n}].s \\ j \in 1..n \qquad f_j = \texttt{mutable} \\ E, x : T, y : T_j \vdash u \,:\, T_j' \,\&\, p_j \end{array}}{E \vdash t.l_j \Leftarrow \varsigma(x, y)u \,:\, \texttt{Unit} \,\&\, \texttt{impure}} \qquad \text{[T-Upd]}$$

$$\frac{}{E \vdash c \,:\, ty(c) \,\&\, \texttt{pure}} \qquad \text{[T-Const]}$$

$$\frac{E \vdash t \,:\, S \,\&\, p \qquad E \vdash T}{E \vdash \langle T \rangle\ t \,:\, T \,\&\, p} \qquad \text{[T-Cast]}$$

$$\frac{E \vdash t \,:\, S \,\&\, q \qquad E \vdash S <: T \qquad q \sqsubseteq p}{E \vdash t \,:\, T \,\&\, p} \qquad \text{[T-Sub]}$$

Type Rules for Types $\boxed{E \vdash T}$

$$\frac{E, x : B \vdash t \,:\, \texttt{Bool} \,\&\, \texttt{pure}}{E \vdash \{x\!:\!B \,|\, t\}} \qquad \text{[T-Base]}$$

$$\frac{\begin{array}{c} T = x\!:\![l_i : f_i \cdot p_i \cdot (y\!:\!T_i \to T_i')\ ^{i \in 1..n}].t \\ S_k = x\!:\![l_i : f_i \cdot p_i \cdot (y\!:\!T_i \to T_i')\ ^{i \in 1..k}].\texttt{true} \\ E, x : S_{j-1} \vdash T_j \quad \forall j \in 1..n \\ E, x : S_{j-1}, y : T_j \vdash T_j' \quad \forall j \in 1..n \\ E, x : S_n \vdash t \,:\, \texttt{Bool} \,\&\, \texttt{pure} \end{array}}{E \vdash T} \qquad \text{[T-ObjTy]}$$

The apparent circularity where the type of $+$ is defined in terms of $+$ itself does not cause any technical difficulties in our development, since the meaning of the refinement predicate is defined in terms of the operational semantics, and hence in terms of the meaning function $[\![+]\!]$.

- The rule [T-Cast] allows a term of inferred type S to be cast to any other (well-formed) type T. Note that this cast may fail at run time.

Figure 4: Subtyping

Subtyping $\boxed{E \vdash S <: T}$

$$\frac{E, x : B \vdash s \Rightarrow t}{E \vdash \{x{:}B \mid s\} <: \{x{:}B \mid t\}} \quad \text{[S-Base]}$$

$$\frac{\begin{array}{l} S = x{:}[l_i : f_i \cdot q_i \cdot M_i \;{}^{i \in 1..n}].s \\ T = x{:}[l_i : f_i \cdot p_i \cdot N_i \;{}^{i \in 1..m}].t \\ m \leq n \qquad q_i \sqsubseteq p_i \quad \forall i \in 1..m \\ f_i = \texttt{final} \\ \quad \Rightarrow \; E, x : S \vdash M_i <: N_i \quad \forall i \in 1..m \\ f_i = \texttt{mutable} \\ \quad \Rightarrow \; E, x : S \vdash M_i = N_i \quad \forall i \in 1..m \\ E, x : S \vdash s \Rightarrow t \end{array}}{E \vdash S <: T} \quad \text{[S-Obj]}$$

Method Subtyping $\boxed{E \vdash M <: N}$

$$\frac{E \vdash T_i <: S_i \qquad E, y : T_i \vdash S'_i <: T'_i}{E \vdash (y{:}S_i \to S'_i) <: (y{:}T_i \to T'_i)} \quad \text{[M-Sub]}$$

Method Equality $\boxed{E \vdash M = N}$

$$\frac{E \vdash M <: N \qquad E \vdash N <: M}{E \vdash M = N} \quad \text{[M-Equal]}$$

Implication $\boxed{E \vdash s \Rightarrow t}$

$$\frac{\begin{array}{l} \forall \theta.\ (E \models \theta \wedge\ (\emptyset, \theta(t)) \longrightarrow^* (\sigma, \texttt{false})) \\ \quad \Rightarrow\ (\emptyset, \theta(s)) \longrightarrow^* (\sigma', \texttt{false}) \end{array}}{E \vdash s \Rightarrow t} \quad \text{[Imp]}$$

Consistent Substitutions $\boxed{E \models \theta}$

$$\frac{}{\emptyset \models \epsilon} \quad \text{[Sub-Empty]}$$

$$\frac{\begin{array}{c} \emptyset \vdash t : T \;\&\; \texttt{pure} \\ E[x := t] \models [y_i := t_i \;{}^{i \in 1..n}] \end{array}}{x : T, E \models [x := t, y_i := t_i \;{}^{i \in 1..n}]} \quad \text{[Sub-Ext]}$$

Well-formed environments $\boxed{\vdash E}$

$$\frac{}{\vdash \emptyset} \quad \text{[WE-Empty]}$$

$$\frac{\vdash E \quad E \vdash T}{\vdash E, x : T} \quad \text{[WE-Ext]}$$

- The rule [T-Sub] allows the inferred type S of a term to be weakened to any super type T.

The judgment $E \vdash T$ checks that the type T is well-formed in environment E.

- The rule [T-Base] states that refinement predicates must have type `Bool` and be pure.
- The rule [T-ObjTy] states that object invariants must have type `Bool` and be pure, and that each method type can only refer to methods declared earlier in the object.

The most interesting part of our type system concerns the rules that define the subtyping judgment $E \vdash S <: T$ in Figure 4. This subtyping judgment is complicated by the expressiveness of refinement predicates and object invariants, and it is defined in terms of the implication judgment $E \vdash s \Rightarrow t$. Essentially, this implication judgment holds if whenever the term t evaluates to `false`, the term s must also evaluate to `false`. More formally, we define a substitution θ (from variables to terms) to be *consistent* with an environment E if θ maps variables to terms in a manner that is consistent with the type bindings in E. The implication judgment $E \vdash s \Rightarrow t$ then holds if for all substitutions θ consistent with E such that $\theta(t)$ evaluates to `false`, it follows that $\theta(s)$ also evaluates to `false`.

Subtyping between refinement types then reduces to an implication judgment between the refinement predicates, via the rule [S-Base]. Subtyping between two object types S and T is covariant on immutable methods and invariant on `mutable` methods. Subtyping also involves checking implication between object invariants. Note that this implication check is performed in an environment where the self-reference variable has type S. This binding allows subtyping to *refactor* type information from method types to the object invariant. For example, suppose:

$$\begin{array}{lcll} S & = & x{:}[m : \{z{:}\texttt{Int} \mid z \geq 0\}].\texttt{true} \\ T & = & x{:}[m : \texttt{Int} \qquad\qquad].(x.m \geq 0) \end{array}$$

Clearly, if the variable x has type S, then the object invariant $(x.m \geq 0)$ returns `true`, or else diverges. Hence:

$$x : S \vdash \texttt{true} \Rightarrow (x.m \geq 0)$$

and so $\emptyset \vdash S <: T$. This subtyping judgment has the effect of refactoring information about the method m from the method type to the object invariant.

6. Hybrid Type Checking

Given that type checking is undecidable, we now present a decidable compilation strategy for Hoop programs. This compilation strategy ensures (by inserting dynamic checks, if necessary) that specifications can never be violated at run time, and it statically rejects clearly ill-typed programs.

The compilation strategy relies on an algorithm to conservatively approximate the logical implication judgment $E \vdash s \Rightarrow t$. The result of this algorithm is denoted as:

$$E \vdash^a_{alg} s \Rightarrow t$$

where the *mode* $a \in \{\surd, \times, ?\}$ indicates whether or not the algorithm is successful at verifying or refuting the given implication. In particular,

- $E \vdash^{\surd}_{alg} s \Rightarrow t$ means the algorithm successfully determines that $E \vdash s \Rightarrow t$.
- $E \vdash^{\times}_{alg} s \Rightarrow t$ means the algorithm determines that the implication does not hold, *i.e.*, $E \not\vdash s \Rightarrow t$.
- $E \vdash^{?}_{alg} s \Rightarrow t$ means the algorithm fails to either prove or disprove that $E \vdash s \Rightarrow t$.

However, we do not require that the algorithm be complete. For example, the trivial algorithm that always returns "?" satisfies our requirements, although it precludes performing any interesting reasoning in the compiler.

Figure 5: Algorithmic Subtyping

Algorithmic Subtyping $\boxed{E \vdash^a_{alg} S <: T}$

$$\frac{E, x : B \vdash^a_{alg} s \Rightarrow t}{E \vdash^a_{alg} \{x{:}B \mid s\} <: \{x{:}B \mid t\}} \quad [\text{AS-}\textsc{Base}]$$

$$\frac{\begin{array}{l} \quad S = x{:}[l_i : f_i \cdot q_i \cdot M_i \ ^{i \in 1..n}].s \\ \quad T = x{:}[l_i : f_i \cdot p_i \cdot N_i \ ^{i \in 1..m}].t \\ \quad m \leq n \qquad q_i \sqsubseteq p_i \quad \forall i \in 1..m \\ f_i = \texttt{final} \\ \quad \Rightarrow E, x : S \vdash^{a_i}_{alg} M_i <: N_i \quad \forall i \in 1..m \\ f_i = \texttt{mutable} \\ \quad \Rightarrow E, x : S \vdash^{a_i}_{alg} M_i = N_i \quad \forall i \in 1..m \\ \quad E, x : S \vdash^a_{alg} s \Rightarrow t \\ \quad a' = a \otimes a_1 \otimes \ldots \otimes a_m \end{array}}{E \vdash^{a'}_{alg} S <: T} \quad [\text{AS-}\textsc{Obj}]$$

Algorithmic Method Subtyping $\boxed{E \vdash^a_{alg} M <: N}$

$$\frac{E \vdash^a_{alg} T_i <: S_i \qquad E, y : T_i \vdash^{a'}_{alg} S'_i <: T'_i}{E \vdash^{(a \otimes a')}_{alg} (y{:}S_i \rightarrow S'_i) <: (y{:}T_i \rightarrow T'_i)} \quad [\text{AM-}\textsc{Sub}]$$

Algorithmic Method Equality $\boxed{E \vdash^a_{alg} M = N}$

$$\frac{E \vdash^a_{alg} M <: N \qquad E \vdash^{a'}_{alg} N <: M}{E \vdash^{(a \otimes a')}_{alg} M = N} \quad [\text{AM-}\textsc{Equal}]$$

Figure 5 defines algorithmic subtyping and algorithmic type equality in terms of this implication algorithm. These rules closely match the corresponding rules in the type system. For example, algorithmic subtyping for base types directly reduces to an algorithmic implication test via [AS-Base]. For subtyping between object types, the rule [AS-Obj] uses the rules [AM-Sub] and [AM-Equal] to ensure the appropriate relationship between corresponding method types, and uses the implication algorithm to check implication of object invariants. As with algorithmic implication, algorithmic subtyping may not always yield a definitive answer. If all sub-tests succeed (or at least one sub-test fails) then the algorithm can conclude with certainty that subtyping does (or does not) hold. In other situations, the result of the whole test is uncertain. We use the 3-valued conjunction operator $\otimes$ to compute the result of a subtype test based on the results of its subtests:

$\otimes$	$\surd$	$\times$	?
$\surd$	$\surd$	$\times$	?
$\times$	$\times$	$\times$	$\times$
?	?	$\times$	?

We now define the hybrid compilation judgment

$$E \vdash s \hookrightarrow t : T \ \& \ p$$

which compiles the term s into the term t, which has type T and purity p. The compilation rules type check the program s and insert additional type casts to compensate for indefinite answers returned by the subtyping algorithm, yielding the compiled program t. Successful compilation does not guarantee that the source program s is well-typed. However,

Figure 6: Compilation Rules

Compilation of Terms $\boxed{E \vdash s \hookrightarrow t : T \ \& \ p}$

$$\frac{(x : T) \in E}{E \vdash x \hookrightarrow x : T \ \& \ \texttt{pure}} \quad [\text{C-}\textsc{Var}]$$

$$\frac{\begin{array}{c} E \vdash S \hookrightarrow T \\ T = x{:}[l_i : f_i \cdot p_i \cdot (y{:}T_i \rightarrow T'_i) \ ^{i \in 1..n}].\texttt{true} \\ f_i = \texttt{mutable} \ \Rightarrow \ p_i = \texttt{impure} \quad \forall i \in 1..n \\ E, x : T, y : T_i \vdash s_i \hookrightarrow t_i \downarrow T'_i \ \& \ p_i \quad \forall i \in 1..n \end{array}}{\begin{array}{l} E \vdash (x.[l_i(y) = s_i \ ^{i \in 1..n}] \text{ as } S) \\ \quad \hookrightarrow (x.[l_i(y) = t_i \ ^{i \in 1..n}] \text{ as } T) : T \ \& \ \texttt{pure} \end{array}} \quad [\text{C-}\textsc{Obj}]$$

$$\frac{\begin{array}{c} E \vdash s_1 \hookrightarrow t_1 : T \ \& \ \texttt{pure} \\ T = x{:}[l_i : f_i \cdot p_i \cdot (y{:}T_i \rightarrow T'_i) \ ^{i \in 1..n}].t' \\ E \vdash s_2 \hookrightarrow t_2 \downarrow T_j[x := t] \ \& \ \texttt{pure} \qquad j \in 1..n \end{array}}{\begin{array}{l} E \vdash s_1.l_j(s_2) \\ \quad \hookrightarrow t_1.l_j(t_2) : T'_j[x := t_1, y := t_2] \ \& \ p_j \end{array}} \quad [\text{C-}\textsc{Sel}]$$

$$\frac{\begin{array}{c} E \vdash s \hookrightarrow t : T \ \& \ p \\ T = x{:}[l_i : f_i \cdot p_i \cdot (y{:}T_i \rightarrow T'_i) \ ^{i \in 1..n}].t' \\ j \in 1..n \qquad f_j = \texttt{mutable} \\ E, x : T, y : T_i \vdash u \hookrightarrow u' \downarrow T'_j \ \& \ p_j \end{array}}{\begin{array}{l} E \vdash (s.l_j \Leftarrow \varsigma(x, y)u) \\ \quad \hookrightarrow (t.l_j \Leftarrow \varsigma(x, y)u') : \texttt{Unit} \ \& \ \texttt{impure} \end{array}} \quad [\text{C-}\textsc{Upd}]$$

$$\frac{}{E \vdash c \hookrightarrow c : ty(c) \ \& \ \texttt{pure}} \quad [\text{C-}\textsc{Const}]$$

$$\frac{E \vdash s \hookrightarrow t : U \ \& \ p \qquad E \vdash S \hookrightarrow T}{E \vdash \langle S \rangle \ s \hookrightarrow \langle T \rangle \ t : T \ \& \ p} \quad [\text{C-}\textsc{Cast}]$$

$$\frac{\begin{array}{c} E \vdash s_1 \hookrightarrow t_1 : U \ \& \ p_1 \\ E \vdash S \hookrightarrow T \\ E, x : U \vdash s_2 \hookrightarrow t_2 \downarrow T \ \& \ p_2 \end{array}}{\begin{array}{l} E \vdash (\texttt{let } x = s_1 \texttt{ in } s_2 \texttt{ as } S) \\ \quad \hookrightarrow (\texttt{let } x = t_1 \texttt{ in } t_2 \texttt{ as } T) : T \ \& \ (p_1 \sqcup p_2) \end{array}} \quad [\text{C-}\textsc{Let}]$$

Compilation of Types $\boxed{E \vdash T \hookrightarrow T'}$

$$\frac{E, x : B \vdash s \hookrightarrow t : \{y{:}\texttt{Bool} \mid u\} \ \& \ \texttt{pure}}{E \vdash \{x{:}B \mid s\} \hookrightarrow \{x{:}B \mid t\}} \quad [\text{C-}\textsc{Base}]$$

$$\frac{\begin{array}{c} S = x{:}[l_i : f_i \cdot p_i \cdot (y{:}S_i \rightarrow S'_i) \ ^{i \in 1..n}].s \\ U_k = x{:}[l_i : f_i \cdot p_i \cdot (y{:}T_i \rightarrow T'_i) \ ^{i \in 1..k}].\texttt{true} \\ T = x{:}[l_i : f_i \cdot p_i \cdot (y{:}T_i \rightarrow T'_i) \ ^{i \in 1..n}].t \\ E, x : U_{j-1} \vdash S_j \hookrightarrow T_j \quad \forall j \in 1..n \\ E, x : U_{j-1}, y : T_j \vdash S'_j \hookrightarrow T'_j \quad \forall j \in 1..n \\ E, x : U_n \vdash s \hookrightarrow t : \{y{:}\texttt{Bool} \mid u\} \ \& \ \texttt{pure} \end{array}}{E \vdash S \hookrightarrow T} \quad [\text{C-}\textsc{ObjTy}]$$

Compilation and Checking $\boxed{E \vdash s \hookrightarrow t : T \ \& \ p}$

$$\frac{E \vdash s \hookrightarrow t : S \ \& \ q \qquad E \vdash^{\surd}_{alg} S <: T \qquad q \sqsubseteq p}{E \vdash s \hookrightarrow t \downarrow T \ \& \ p} \quad [\text{CC-}\textsc{Ok}]$$

$$\frac{E \vdash s \hookrightarrow t : S \ \& \ q \qquad E \vdash^{?}_{alg} S <: T \qquad q \sqsubseteq p}{E \vdash s \hookrightarrow (\langle T \rangle \ t) \downarrow T \ \& \ p} \quad [\text{CC-}\textsc{Chk}]$$

if s is well-typed, then all dynamic casts inserted into t will succeed. If s is not well-typed, then execution of t may potentially halt due to the failure of a compiler-inserted cast. We extend compilation to types as well, since they contain terms.

The full set of compilation rules is shown in Figure 6. Many of these rules are straightforward. Variable references and constants never need additional casts, as evidenced by rules [C-Var] and [C-Const]. To compile an object $(x.[l_i(y) = s_i \ ^{i\in 1..n}]$ as $S)$, the rule [C-Obj] first compiles the object type S into type

$$T = x\!:\![l_i : f_i \cdot p_i \cdot (y\!:\!T_i \to T_i') \ ^{i\in 1..n}].\texttt{true}$$

and then compiles each method body s_i into t_i, which must have type $y\!:\!T_i \to T_i'$.

We use the compilation and checking judgment

$$E \vdash s \hookrightarrow t \downarrow T \;\&\; p$$

to both compile s into t and to ensure that t has type T (and purity p). The rules defining this judgment exhibit the key ideas behind hybrid type checking for objects. After compiling the term s into t (with type S), the subtyping algorithm $E \vdash^a_{alg} S <: T$ is used to check if S is a subtype of T.

- If the subtyping algorithm succeeds (*i.e.*, $E \vdash^{\surd}_{alg} S <: T$), then no casts are needed: see [CC-Ok].
- If the subtyping algorithm disproves this subtyping requirement (*i.e.*, $E \vdash^{\times}_{alg} S <: T$), the program fails to compile, since no compilation rule is applicable.
- If the subtyping algorithm cannot either prove or refute this subtyping requirement (*i.e.*, $E \vdash^{?}_{alg} S <: T$), then a dynamic cast from S to T is inserted around t via the rule [CC-Chk].

Ideally, the subtyping algorithm will be sufficiently complete to handle most cases definitively and casts will be inserted rarely. However, this crucial ability to add casts permits our compiler to support an unusually expressive object specification language.

The compiler compiles types wherever they are found in the source program (*i.e.*, in [C-Cast] and [C-Obj]), to ensure that all types mentioned in the compiled program are well-formed, and that all environments constructed during compilation are well-formed.

7. Correctness

In this section, we establish important correctness properties of the Hoop type system and the hybrid compilation algorithm.

7.1 Correctness of the Type System

We first extend the type system to run-time states, which contain an object store as well as object addresses and object views. In particular, we extend each typing judgment to include an object store σ, resulting in the extended judgment forms:

$$\begin{array}{lll} \sigma; E \vdash t : T \;\&\; p & \sigma; E \vdash T & \sigma; E \vdash S <: T \\ \sigma; E \vdash M = N & \sigma; E \vdash M <: N & \\ \sigma; E \vdash s \Rightarrow t & \sigma; E \models \theta & \end{array}$$

Most of the extended type rules are identical to the original rules, except that they pass the additional store argument σ to their antecedents. The only exception is the following rule [Imp], which uses the store σ when evaluating implication predicates:

$$\frac{\begin{array}{c}\forall\theta.\ (\sigma; E \models \theta \wedge\ (\sigma, \theta(t)) \longrightarrow^* (\sigma', \texttt{false})) \\ \Rightarrow\ (\sigma, \theta(s)) \longrightarrow^* (\sigma'', \texttt{false})\end{array}}{\sigma; E \vdash s \Rightarrow t} \quad [\text{Imp}]$$

We introduce two new rules for checking object addresses and object views:

$$\frac{\sigma(a) = x.[\ldots] \text{ as } T}{\sigma; E \vdash a \,:\, T \;\&\; \texttt{pure}} \quad [\text{T-Obj-Ref}]$$

$$\frac{\begin{array}{c}\sigma(a) = x.[l_i(y) = t_i \ ^{i\in 1..n}] \text{ as } S \\ S = x\!:\![l_i : f_i \cdot q_i \cdot M_i \ ^{i\in 1..n}].s \\ T = x\!:\![l_i : f_i \cdot p_i \cdot N_i \ ^{i\in 1..m}].t \\ T' = x\!:\![l_i : f_i \cdot p_i \cdot N_i \ ^{i\in 1..m}].\texttt{true} \\ m \le n \qquad q_i \sqsubseteq p_i \quad \forall i \in 1..m \\ (\sigma, t[x := (a \texttt{ view } T')]) \longrightarrow^* (\sigma', \texttt{true})\end{array}}{\sigma; E \vdash a \texttt{ view } T \,:\, T \;\&\; \texttt{pure}} \quad [\text{T-Obj-View}]$$

Finally, we add rules for typing stores and states:

$$\frac{\begin{array}{c}\text{dom}(\sigma) = \{a_1, \ldots, a_n\} \\ \sigma(a_i) = x.[\ldots] \text{ as } S_i \qquad \forall i \in 1..n \\ \sigma; \emptyset \vdash \sigma(a_i) \,:\, S_i \;\&\; \texttt{pure}\end{array}}{\vdash \sigma} \quad [\text{T-Store}]$$

$$\frac{\vdash \sigma \qquad \sigma; \emptyset \vdash t \,:\, T \;\&\; p}{\vdash (\sigma, t) : T} \quad [\text{T-State}]$$

We assume that the type of each primitive operation is consistent with its operational behavior.

Assumption 1 (Types of Primitives). *If* $\vdash (\sigma, c.l(v)) : T$ *and* $[\![c]\!](l, v, \sigma) = v'$ *then* $\vdash (\sigma, v') : T$.

The extended type system then satisfies the preservation or subject reduction property [41]. The type system also satisfies the progress property (*i.e.*, the evaluation of well-typed programs does not halt prematurely), with the caveat that type casts may fail. A state $(\sigma, \mathcal{E}[\langle T\rangle\ v])$ has a *failed cast* if it cannot be reduced via the rules [E-Cast-Base], [E-Cast-Obj], or [E-Cast-View]. Both theorems follow by induction over typing derivations.

Theorem 2 (Preservation). *If* $\vdash (\sigma, s) : T$ *and* $(\sigma, s) \longrightarrow (\sigma', t)$ *then* $\vdash (\sigma', t) : T$.

Theorem 3 (Progress). *All closed well-typed normal forms are values or contain a failed cast.*

7.2 Correctness of Hybrid Type System

We now describe the correctness properties of our hybrid type system. We assume that the implication algorithm is a sound approximation of the implication judgment:

Assumption 4 (Soundness of $E \vdash^a_{alg} s \Rightarrow t$). *Suppose* $\vdash E$.

1. *If* $E \vdash^{\surd}_{alg} s \Rightarrow t$ *then* $E \vdash s \Rightarrow t$.
2. *If* $E \vdash^{\times}_{alg} s \Rightarrow t$ *then* $E \not\vdash s \Rightarrow t$.

Given this assumption, our subtyping algorithm is sound:

Lemma 5 (Soundness of $E \vdash^a_{alg} S <: T$). *Suppose* $\vdash E$.

1. *If* $E \vdash^{\surd}_{alg} S <: T$ *then* $E \vdash S <: T$.

2. If $E \vdash^{\times}_{alg} S <: T$ then $E \not\vdash S <: T$.

Proof: By induction over algorithmic subtyping derivations. □

Also, the compilation algorithm inserts sufficient dynamic type casts to compensate for the inherent imprecisions of the subtyping algorithm, which ensures that compiled programs are always well-typed.

Theorem 6 (Compiled Programs Are Well-typed). *Suppose $\vdash E$.*

1. If $E \vdash s \hookrightarrow t : T \,\&\, p$ then $E \vdash t : T \,\&\, p$.

2. If $E \vdash s \hookrightarrow t \downarrow T \,\&\, p$ and $E \vdash T$ then $E \vdash t : T \,\&\, p$.

3. If $E \vdash S \hookrightarrow T$ then $E \vdash T$.

Proof: By induction over compilation derivations. □

An immediate consequence of this lemma, when combined with Theorem 3 (Progress), is that compiled programs only halt prematurely due to failed casts. Such casts may be explicit in the original program or implicitly inserted by the compiler. Thus, for some subtle specification violations, hybrid type checking may only catch these errors via cast failures at run time. However, we argue that this precise hybrid approach may be superior to using a coarser type language that could not express subtle specifications at all.

8. Related Work

Much prior work has focused on dynamic checking of expressive specifications, or *contracts* [29, 11, 24, 16, 20, 26, 34, 21]. An entire design philosophy, *Contract Oriented Design*, has been based on dynamically-checked specifications. Hybrid type checking extends these prior purely-dynamic approaches by verifying (or detecting violations of) expressive specifications statically, wherever possible.

The programming language Eiffel [29] supports a notion of hybrid specifications by providing both statically-checked types as well as dynamically-checked contracts. Having separate (static and dynamic) specification languages is awkward, since it requires the programmer to factor each specification into its static and dynamic components. Furthermore, the specification may need to be manually refactored to exploit improvements in static checking technology.

Our work shares similar motivations with, and is partly inspired by, recent work on advanced type systems, including work on refinement types [15, 28, 8] and practical dependent types [43, 42]. However, the requirement for full static decidability limits the expressiveness of these prior systems. Hybrid type checking side-steps these decidability difficulties by being willing to check correctness properties dynamically when necessary.

The static checking tool ESC/Java [14] supports expressive JML specifications [24] and leverages powerful automatic theorem proving techniques. However, ESC/Java's underlying theorem prover, Simplify [9], does not distinguish between failing to prove a theorem and finding a counterexample that actually refutes the theorem. Consequently, ESC/Java may produce error messages that are caused by limitations in its theorem prover. In contrast, hybrid type checking only produces error messages for programs it can *prove* are ill-typed.

Abadi and Leino [3] developed and proved the soundness of a Hoare-style logic for reasoning about pre- and post-conditions in an imperative object language. As with ESC/Java, this logic reduces program correctness to theorem proving, which is undecidable. Others have explored various semantic models [36] and extensions [19] to the Abadi-Leino system. Extending our work to support an imperative object language would require studying a number of additional verification issues, such as frame conditions [6] and aliasing/ownership [7, 31], that arise in an imperative setting.

There have been a number of projects focused on refinement of specifications and subtyping. These include the work of Liskov and Wing [25], Leavens [23], Lano and Haughton [22], and others. Our notion of subtyping is simpler than many of these studies, and does not, for example, include data abstraction. One interesting avenue for future work is to explore these richer notions of subtyping and abstraction in the context of hybrid type checking.

The limitations of purely-static and purely-dynamic approaches have also motivated other work on hybrid analyses. For example, CCured [30] is a hybrid analysis for preventing array bounds violations. Unlike our proposed approach, it does not detect errors statically - instead, the static analysis is used to optimize the run-time analysis. Specialized hybrid analyses have been proposed for other problems as well, such as race condition checking [39, 32, 4].

Recently, Ou et al presented a system leveraging dependent types and run-time checks [33], although their system does not include objects. In contrast to Hoop, their type system is decidable and leverages dynamic checks to reduce the need for precise type annotations in explicitly labeled regions of programs.

The interaction between static typing and dynamic checks has been previously studied in context of flexible type systems with types such as the type `Dynamic`, which can be converted to other types [2]. Quasi-static typing [38] automatically inserts the necessary coercions, in a manner similar to soft typing [27, 40, 5, 13]. This work is intended to support looser type specifications. In contrast, our work uses similar, automatically-inserted casts to support more precise type specifications. An interesting avenue for further exploration is the combination of both approaches to support a very large range of specifications, from `Dynamic` (no static type information) at one extreme to very precise hybrid-checked specifications at the other.

9. Conclusions and Future Work

This paper presents a hybrid type system that permits programmers to specify object interfaces precisely, using object invariants and type refinements, and to delegate to the hybrid compiler decisions regarding which parts of these specifications can be checked statically, and which must be checked dynamically. A number of issues remain for future work, including experimental validation.

The desire for static decidability has often constrained traditional object type systems. We believe that hybrid type checking may facilitate the design of programming languages around expressive type systems that balance safety with flexibility, and whose undecidability need not limit their practicality.

Acknowledgments This work was supported by the National Science Foundation under Grants CCR-0341179 and CCR-0341387. We thank Martín Abadi for helpful comments on this work.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag, 1996.

[2] M. Abadi, L. Cardelli, B. Pierce, and G. Plotkin. Dynamic typing in a statically-typed language. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 213–227, 1989.

[3] M. Abadi and K. R. M. Leino. A logic of object-oriented programs. In *Verification: Theory and Practice*, pages 11–41, 2003.

[4] R. Agarwal and S. D. Stoller. Type inference for parameterized race-free Java. In *Proceedings of the Conference on Verification, Model Checking, and Abstract Interpretation*, pages 149–160, 2004.

[5] A. Aiken, E. L. Wimmers, and T. K. Lakshman. Soft typing with conditional types. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 163–173, 1994.

[6] A. Borgida, J. Mylopoulos, and R. Reiter. On the frame problem in procedure specifications. *IEEE Trans. Software Eng.*, 21(10):785–798, 1995.

[7] J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *Proceedings of the European Conference on Object Oriented Programming*, pages 2–27, 2001.

[8] R. Davies and F. Pfenning. Intersection types and computational effects. In *Proceedings of the ACM International Conference on Functional Programming*, pages 198–208, 2000.

[9] D. Detlefs, G. Nelson, and J. B. Saxe. Simplify: a theorem prover for program checking. *J. ACM*, 52(3):365–473, 2005.

[10] ECMA. Standard ECMA-334: C# Language Specification (3rd edition), 2005. Available on the web as `http://www.ecma-international.org/publications/-files/ecma-st/Ecma-334.pdf`.

[11] R. B. Findler and M. Felleisen. Contracts for higher-order functions. In *Proceedings of the International Conference on Functional Programming*, pages 48–59, 2002.

[12] C. Flanagan. Hybrid type checking. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, 2006.

[13] C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Finding bugs in the web of program invariants. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 23–32, 1996.

[14] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J. B. Saxe, and R. Stata. Extended static checking for Java. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 234–245, 2002.

[15] T. Freeman and F. Pfenning. Refinement types for ML. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 268–277, 1991.

[16] B. Gomes, D. Stoutamire, B. Vaysman, and H. Klawitter. A language manual for Sather 1.1, 1996.

[17] J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification (3rd Edition)*. Addison-Wesley, 2005.

[18] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Practical hybrid checking for expressive types and specifications, extended report. `http://www.soe.ucsc.edu/~cormac/papers/sage-full.ps`, to appear, 2005.

[19] M. Hofmann and F. Tang. Implementing a program logic of objects in a higher-order logic theorem prover. In *Proceedings of the Conference on Theorem Proving in Higher Order Logics*, pages 268–282, 2000.

[20] R. C. Holt and J. R. Cordy. The Turing programming language. *Communications of the ACM*, 31:1310–1424, 1988.

[21] M. Kölling and J. Rosenberg. Blue: Language specification, version 0.94, 1997.

[22] K. Lano and H. P. Haughton. Reasoning and refinement in object-oriented specification languages. In *Proceedings of the European Conference on Object Oriented Programming*, pages 78–97, 1992.

[23] G. T. Leavens. *Reasoning about Object-Oriented Programs that Use Subtypes*. PhD thesis, Massachusetts Institute of Technology, 1989.

[24] G. T. Leavens and Y. Cheon. Design by contract with JML, 2005. avaiable at `http://www.cs.iastate.edu/~leavens/JML/`.

[25] B. Liskov and J. M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, 1994.

[26] D. Luckham. Programming with specifications. *Texts and Monographs in Computer Science*, 1990.

[27] M. Fagan. *Soft Typing*. PhD thesis, Rice University, 1990.

[28] Y. Mandelbaum, D. Walker, and R. Harper. An effective theory of type refinements. In *Proceedings of the International Conference on Functional Programming*, pages 213–225, 2003.

[29] B. Meyer. *Object-oriented Software Construction*. Prentice Hall, 1988.

[30] G. C. Necula, S. McPeak, and W. Weimer. CCured: type-safe retrofitting of legacy code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 128–139, 2002.

[31] J. Noble, J. Vitek, and J. Potter. Flexible alias protection. In *Proceedings of the European Conference on Object Oriented Programming*, pages 158–185, 1998.

[32] R. O'Callahan and J.-D. Choi. Hybrid dynamic data race detection. In *ACM Symposium on Principles and Practice of Parallel Programming*, pages 167–178, 2003.

[33] X. Ou, G. Tan, Y. Mandelbaum, and D. Walker. Dynamic typing with dependent types. In *IFIP International Conference on Theoretical Computer Science*, pages 437–450, 2004.

[34] D. L. Parnas. A technique for software module specification with examples. *Communications of the ACM*, 15(5):330–336, 1972.

[35] D. Rémy and J. Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998.

[36] B. Reus and J. Schwinghammer. Denotational semantics for Abadi and Leino's logic of objects. In *Proceedings of the European Symposium on Programming*, pages 263–278, 2005.

[37] D. S. Rosenblum. A practical approach to programming with assertions. *IEEE Transactions on Software Engineering*, 21(1):19–31, 1995.

[38] S. Thatte. Quasi-static typing. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 367–381, 1990.

[39] C. von Praun and T. Gross. Object race detection. In *Proceedings of the ACM Conference on Object-Oriented Programming, Systems, Languages and Applications*, pages 70–82, 2001.

[40] A. Wright and R. Cartwright. A practical soft type system for scheme. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, pages 250–262, 1994.

[41] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Info. Comput.*, 115(1):38–94, 1994.

[42] H. Xi. Imperative programming with dependent types. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 375–387, 2000.

[43] H. Xi and F. Pfenning. Dependent types in practical programming. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 214–227, 1999.