Hybrid Type Checking

KENNETH KNOWLES CORMAC FLANAGAN
University of California at Santa Cruz

Traditional static type systems are effective for verifying basic interface specifications. Dynamically-checked contracts support more precise specifications, but these are not checked until run time, resulting in incomplete detection of defects. Hybrid type checking is a synthesis of these two approaches that enforces precise interface specifications, via static analysis where possible, but also via dynamic checks where necessary. This paper explores the key ideas and implications of hybrid type checking, in the context of the λ-calculus extended with *contract types*, *i.e.*, with dependent function types and with arbitrary refinements of base types.

Categories and Subject Descriptors: D.3.1 [**Programming Languages: Formal Definitions and Theory**]: specification and verification

General Terms: Languages, Theory, Verification

Additional Key Words and Phrases: Type systems, contracts, static checking, dynamic checking

1. MOTIVATION

The construction of reliable software is notoriously difficult, in part because programmers typically work in the context of a large collection of APIs whose behavior is only informally and imprecisely specified and understood. Techniques for specifying and verifying software interfaces have been the focus of much prior work.

Static type systems have proven to be effective and practical tools for verifying basic structural specifications. There are important specifications, however, that cannot be expressed with structural types. Ongoing research on more powerful type systems (*e.g.*, [Freeman and Pfenning 1991; Xi and Pfenning 1999; Xi 2000; Davies and Pfenning 2000; Mandelbaum et al. 2003]) attempts to overcome some of these restrictions via advanced features such as dependent and refinement types that can express more logical aspects of interface specifications. Yet these systems are designed to be *statically type safe*, and so the specification language is intentionally restricted to ensure that specifications can always be checked statically.

In contrast, *dynamic contract checking* [D. L. Parnas 1972; Meyer 1988; Holt and Cordy 1988; Luckham 1990; Gomes et al. 1996; Kölling and Rosenberg 1997; Findler and Felleisen 2002; Leavens and Cheon 2005] provides a simple method for checking and enforcing executable specifications at run-time. Dynamic checking can easily support precise specifications, such as:

- Subranges: The function `printDigit` requires an integer in the range [0,9].
- Aliasing restrictions: The function `swap` requires that its arguments are distinct reference cells.
- Ordering restrictions: The function `binarySearch` requires that its argument is a sorted array.
- Size specifications: The function `serializeMatrix` takes as input a matrix of size n by m, and returns a one-dimensional array of size $n \times m$.

- Arbitrary predicates: an interpreter (or code generator) for a typed language (or intermediate representation [Tarditi et al. 1996]) might naturally require that its input be well-typed, *i.e.*, that it satisfies the predicate `wellTyped` : `Expr` $\rightarrow$ `Bool`.

However, dynamic checking suffers from two limitations. First, it consumes cycles that could otherwise perform useful computation. More seriously, dynamic checking provides only limited coverage – specifications are only checked on data values and code paths of actual executions. Thus, dynamic checking often results in incomplete and late (possibly post-deployment) detection of defects.

Thus, the twin goals of *complete checking* and *expressive specifications* appear somewhat incompatible.[1] Static type checking focuses on complete checking of restricted specifications. Dynamic checking focuses on incomplete checking of expressive specifications. Neither approach in isolation provides an entirely satisfactory solution for enforcing precise interface specifications.

In this paper, we describe an approach for validating precise interface specifications using a synthesis of static and dynamic techniques. By checking correctness properties and detecting defects statically (whenever possible) and dynamically (only when necessary), this approach of *hybrid type checking* attempts to combine the benefits of prior purely-static and purely-dynamic approaches.

We illustrate the key idea of hybrid type checking by considering the type rule for function application:

$$\frac{E \vdash t_1 : T \rightarrow T' \qquad E \vdash t_2 : S \qquad E \vdash S <: T}{E \vdash (t_1\ t_2) : T'}$$

The antecedent $E \vdash S <: T$ checks compatibility of the actual and formal parameter types. If the type checker can prove this subtyping relation, then this application is well-typed. Conversely, if the type checker can prove that this subtyping relation does not hold, then the program is rejected. In a conventional, decidable type system, one of these two cases always holds.

However, once we consider expressive type languages that are not statically decidable, the type checker may encounter situations where its algorithms can neither prove nor refute the subtype judgment $E \vdash S <: T$ (particularly within the time bounds imposed by interactive compilation). A fundamental question in the development of expressive type systems is how to deal with such situations where the compiler cannot statically classify the program as either ill-typed or well-typed:

- *Statically rejecting* such programs would cause the compiler to reject some programs that, on deeper analysis, could be shown to be well-typed. This approach seems a little too brittle for use in practice, since it would be difficult to predict which programs the compiler would accept.
- *Statically accepting* such programs (based on the optimistic assumption that the unproven subtype relations actually hold) may result in specifications being violated at run time, which is undesirable.

[1]Complete checking of expressive specifications could be achieved by requiring that each program be accompanied by a proof (perhaps expressed as type annotations) that the program satisfies its specification, but manually or interactively writing such proofs is currently rather heavyweight for widespread use.

Ill-typed programs		**Well-typed programs**	
Clearly ill-typed	Subtle programs		Clearly well-typed
Rejected by type checker	Accepted with casts		Accepted without casts
	Casts may fail	Casts never fail	

Fig. 1. Hybrid type checking on various programs.

Hence, a promising approach is for the compiler to accept such programs on a provisional basis, but to insert sufficient dynamic checks to ensure that specification violations never occur at run time. Of course, checking that $E \vdash S <: T$ at run time is still a difficult problem and would violate the principle of *phase distinction* [Cardelli 1988a]. Instead, our hybrid type checking approach transforms the above application into the code

$$t_1\ (\langle T \triangleleft S\rangle\ t_2)$$

where the additional *typecast* or *coercion* $\langle T \triangleleft S\rangle\ t_2$ dynamically checks that the value produced by t_2 is in the domain type T. Note that hybrid type checking supports precise types, and T could in fact specify a detailed precondition of the function, for example, that it only accepts prime numbers. In this case, the run-time cast would involve performing a primality check.

The behavior of hybrid type checking on various kinds of programs is illustrated in Figure 1. Although every program can be classified as either ill-typed or well-typed, for expressive type systems it is not always possible to make this classification statically. However, the compiler can still identify some (hopefully many) clearly ill-typed programs, which are rejected, and similarly can identify some clearly well-typed programs, which are accepted unchanged.

For the remaining *subtle* programs, dynamic type casts are inserted to check any unverified correctness properties at run time. If the original program is actually well-typed, these casts are redundant and will never fail. Conversely, if the original program is ill-typed in a subtle manner that cannot easily be detected at compile time, the inserted casts may fail. As static analysis technology improves, we expect that the category of subtle programs in Figure 1 will shrink, as more ill-typed programs are rejected and more well-typed programs are fully verified at compile time.

Hybrid type checking provides several desirable characteristics:

(1) It supports precise interface specifications, which facilitate modular development of reliable software.
(2) As many defects as possible and practical are detected at compile time (and we expect this set will increase as static analysis technology evolves).
(3) All well-typed programs are accepted by the checker.
(4) 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 specification violations are always detected, either statically or dynamically.

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

(6) If the source program is well-typed, then the inserted casts are guaranteed to succeed, and so the source and output programs are behaviorally equivalent.

Our proposed specifications extend traditional static types, and so we view hybrid type checking as an extension of traditional static type checking. In particular, hybrid type checking supports precise specifications while preserving a key benefit of static type systems; namely, the ability to detect simple syntactic errors at compile time.

Hybrid type checking may facilitiate the evolution and adoption of advanced static analyses, by allowing software engineers to experiment with sophisticated specification strategies that cannot (yet) be verified statically. Such experiments can then motivate and direct static analysis research. In particular, if a hybrid type checker fails to decide (*i.e.*, verify or refute) a subtyping query, it could send that query back to the compiler writer. Similarly, if a hybrid-typed program fails an inserted cast $\langle T \triangleleft S\rangle\ v$, the value v is a witness that refutes an undecided subtyping query $S <: T$, and such witnesses could also be sent back to the compiler writer. This information would provide concrete and quantifiable motivation for subsequent improvements in the type checker's analysis.

Indeed, just as different compilers for the same language may yield object code of varying quality, we might imagine a variety of hybrid type checkers with different trade-offs between static and dynamic checks (and between static and dynamic error messages). Fast interactive hybrid compilers might perform only limited static analysis to detect obvious type errors, while production compilers could perform deeper analyses to detect more defects statically and to generate improved code with fewer dynamic checks.

Hybrid type checking is inspired by prior work on soft typing [Fagan, M. 1990; Wright and Cartwright 1994; Aiken et al. 1994; Flanagan et al. 1996], but it extends soft typing by rejecting many ill-typed programs, in the spirit of static type checkers. The interaction between static typing and dynamic checks has also been studied in the context of type systems with the type `Dynamic` [Abadi, M., L. Cardelli, B. Pierce, and G. Plotkin 1989; Thatte, S. 1990], and in systems that combine dynamic checks with dependent types [Ou et al. 2004]. Hybrid type checking extends these ideas to support more precise specifications.

The general approach of hybrid type checking appears applicable to a variety of programming languages and specification languages. In this paper, we develop this approach for a fairly expressive dependent type system that is statically undecidable. Specifically, we work with an extension of the λ-calculus with *contract types*, that is, with dependent function types and arbitrary refinements of base types.

The presentation of our results proceeds as follows. Section 2 introduces our core calculus, together with its operational semantics and (undecidable) type system. Section 3 presents a hybrid type checking algorithm for this language, and Section 4 illustrates this algorithm on an example program. Section 5 formalizes the idea of *closing substitutions*, a key technical notion in our formalism. Section 6 verifies correctness properties of the type system and of the hybrid type checking algorithm. Section 7 formally compares the static and hybrid approaches to type checking.

Section 8 relates our approach to other work is this area, and Section 9 outlines opportunities for future research.

This paper is an extended version of an earlier conference paper [Flanagan 2006] that revises some problematic aspects and that provides simpler proofs of correctness. In particular, our earlier type system was defined via a collection of mutually-recursive rules for the typing, subtyping, and implication relations, where the definition of the implication relation refers to the typing relation in a negative position. Thus, standard monotonicity arguments are not applicable, and so it is not obvious what non-trivial type systems satisfy these rules. (We note that [Ou et al. 2004] and [Gronski et al. 2006] use an alternative approach of axiomatizing the implication relation, but it is still not clear which relations satisfy those axioms.)

This paper circumvents these issues by leveraging a denotational interpretation of contract types as dynamically-checked contracts on top of the simply-typed lambda calculus. This approach allows us to define closing substitutions and the implication relations in a non-circular manner (Section 5), which then provides a solid foundation for the remainder of the contract type system.

In addition, this paper proves that compilation preserves equivalence via the proof technique of *logical relations*, which provides a more elegant approach than the complex and somewhat brittle bisimulation relation of [Flanagan 2006]. Finally, [Flanagan 2006] defined the small-step evaluation relation in terms of its own reflexive-transitive closure. We present a cleaner operational semantics by adding a syntactic form for a run-time check "in progress", as described in the following section.

2. THE LANGUAGE λ^H

This section introduces an extension of the λ-calculus with contract types, *i.e.*, with dependent function types and with precise (and hence undecidable) refinement types. We refer to this language as λ^H.

2.1 Syntax of λ^H

The syntax of λ^H is summarized in Figure 2. Terms include variables, constants, functions, applications, and casts. The cast $\langle T \triangleleft S \rangle$ is a function of type $S \to T$, and dynamically checks if its argument, statically known to be of type S, is also of type T (in a manner similar to coercions [Thatte, S. 1990], contracts [Findler 2002; Findler and Felleisen 2002], and to type casts in languages such as Java [Gosling et al. 1996]). The cast-in-progress construct $\langle T, t, c \rangle$ is introduced during program evaluation, as discussed in Section 2.2 below.

The λ^H type language includes dependent function types [Cardelli 1988b], for which we use the syntax $x : S \to T$ of Cayenne [Augustsson 1998]. Here, S is the domain type of the function and the formal parameter x may occur in the range type T. We omit x if it does not occur free in T, yielding the standard function type syntax $S \to T$.

We use B to range over base types, which includes at least `Bool` and `Int`. As in many languages, these base types are fairly coarse and cannot, for example, denote integer subranges. To overcome this limitation, we introduce *refinement types* of

Figure 2: Syntax

$v ::=$	*Values:*
c	constant
$\lambda x\!:\!S.\,t$	abstraction
$\langle T \triangleleft S\rangle$	type cast
$s, t ::=$	*Terms:*
v	value
x	variable
$t\ t$	application
$\langle T, t, c\rangle$	cast in progress
$S, T ::=$	*Types:*
$x\!:\!S \rightarrow T$	dependent function type
$\{x\!:\!B \mid t\}$	refinement type
$B ::=$	*Base types:*
`Int`	base type of integers
`Bool`	base type of booleans
$E ::=$	*Environments:*
$\emptyset$	empty environment
$E, x : T$	environment extension

the form

$$\{x\!:\!B \mid t\}$$

Here, the variable x (of type B) can occur within the boolean term or *predicate* t. This refinement type denotes the set of constants c of type B that satisfy this predicate, *i.e.*, for which the term $[x \mapsto c]\,t$ evaluates to true, where $[x \mapsto c]$ is the substitution function that replaces x by c in its argument. Thus, $\{x\!:\!B \mid t\}$ denotes a subtype of B, and we use a base type B as an abbreviation for the trivial refinement type $\{x\!:\!B \mid \texttt{true}\}$.

These refinement types are inspired by prior work on decidable refinement type systems [Freeman and Pfenning 1991; Xi and Pfenning 1999; Xi 2000; Davies and Pfenning 2000; Mandelbaum et al. 2003; Ou et al. 2004]. However, since refinement predicates are arbitrary boolean expressions, every computable subset of the integers is actually a λ^H type. Not surprisingly, this expressive power causes type checking to become 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 predicates, which is clearly undecidable. These decidability difficulties are circumvented by our hybrid type checking algorithm, which we describe in Section 3.

The type of each constant is defined by the following function ty : $Constant \rightarrow$

Type, and the set *Constant* is implicitly defined as the domain of this mapping.

$$\begin{array}{rcl}
\texttt{true} & : & \{b{:}\texttt{Bool} \mid b \Leftrightarrow \texttt{true}\} \\
\texttt{false} & : & \{b{:}\texttt{Bool} \mid b \Leftrightarrow \texttt{false}\} \\
\Leftrightarrow & : & b_1{:}\texttt{Bool} \to b_2{:}\texttt{Bool} \to \{b{:}\texttt{Bool} \mid b \Leftrightarrow (b_1 \Leftrightarrow b_2)\} \\
\texttt{not} & : & b{:}\texttt{Bool} \to \{b'{:}\texttt{Bool} \mid b \Leftrightarrow \texttt{not}\ b\} \\
n & : & \{m{:}\texttt{Int} \mid m = n\} \\
+ & : & n{:}\texttt{Int} \to m{:}\texttt{Int} \to \{z{:}\texttt{Int} \mid z = n + m\} \\
+_n & : & m{:}\texttt{Int} \to \{z{:}\texttt{Int} \mid z = n + m\} \\
/ & : & n{:}\texttt{Int} \to m{:}\{z{:}\texttt{Int} \mid z \neq 0\} \to \{z{:}\texttt{Int} \mid z = n + m\} \\
= & : & n{:}\texttt{Int} \to m{:}\texttt{Int} \to \{b{:}\texttt{Bool} \mid b \Leftrightarrow (n = m)\} \\
\texttt{if}_T & : & \texttt{Bool} \to T \to T \to T \\
\texttt{fix}_T & : & (T \to T) \to T
\end{array}$$

A *basic constant* is a constant whose type is a refinement type (i.e. not a function type). Each basic constant is assigned a singleton type that denotes exactly that constant. For example, the type of an integer n denotes the singleton set $\{n\}$.

A *primitive function* is a constant of function type. For clarity, we use infix syntax for applications of some primitive functions (*e.g.*, $+$, $=$, $\Leftrightarrow$). The types for primitive functions are quite precise. For example, the type for the primitive function $+$:

$$n{:}\texttt{Int} \to m{:}\texttt{Int} \to \{z{:}\texttt{Int} \mid z = n + m\}$$

exactly specifies that this function performs addition. That is, the term $n + m$ has the type $\{z{:}\texttt{Int} \mid z = n + m\}$ denoting the singleton set $\{n + m\}$. Note that even though the type of "+" is defined in terms of "+" itself, this does not cause any problems in our technical development, since the semantics of refinement predicates is defined in terms of the operational semantics.

The constant $\texttt{fix}_T$ is the fixpoint constructor of type T, and enables the definition of recursive functions. For example, the factorial function can be defined as:

```
fix_{Int→Int}
  λf:(Int → Int).
    λn:Int.
      if_{Int} (n = 0)
               1
               (n ∗ (f (n − 1)))
```

Contract types can express many precise specifications, such as the following (where we assume that `Unit`, `Array`, and `RefInt` are additional base types, and the primitive function `sorted` : `Array` → `Bool` identifies sorted arrays.)

- `printDigit` : $\{x{:}\texttt{Int} \mid 0 \leq x \wedge x \leq 9\} \to \texttt{Unit}$.
- `swap` : $x{:}\texttt{RefInt} \to \{y{:}\texttt{RefInt} \mid x \neq y\} \to \texttt{Unit}$.
- `binarySearch` : $\{a{:}\texttt{Array} \mid \texttt{sorted}\ a\} \to \texttt{Int} \to \texttt{Bool}$.

2.2 Operational Semantics of λ^H

We next describe the run-time behavior of λ^H terms, since the semantics of the type language depends on the operational semantics of terms. The relation $s \leadsto t$

Figure 3: Evaluation Rules

Redex Evaluation $\boxed{s \longrightarrow t}$

$$(\lambda x{:}S.\, t)\ s \ \longrightarrow\ [x \mapsto s]\, t \qquad \text{[E-BETA]}$$

$$c\ v \ \longrightarrow\ \delta(c, v) \qquad \text{[E-PRIM]}$$

$$\langle x{:}T_1 \to T_2 \lhd x{:}S_1 \to S_2\rangle\ v \ \longrightarrow\ \lambda x{:}T_1.\, \langle T_2 \lhd [x \mapsto \langle S_1 \lhd T_1\rangle\ x]\, S_2\rangle\ (v\ (\langle S_1 \lhd T_1\rangle\ x)) \qquad \text{[E-CAST-FN]}$$

$$\langle \{x{:}B \mid t\} \lhd \{x{:}B \mid s\}\rangle\ c \ \longrightarrow\ \langle \{x{:}B \mid t\}, [x \mapsto c]\, t, c\rangle \qquad \text{[E-CAST-BEGIN]}$$

$$\langle \{x{:}B \mid s\}, \mathtt{true}, c\rangle \ \longrightarrow\ c \qquad \text{[E-CAST-END]}$$

Contextual Evaluation $\boxed{s \leadsto t}$

$$\mathcal{C}[s] \ \leadsto\ \mathcal{C}[t] \quad \text{if } s \longrightarrow t \qquad \text{[E-COMPAT]}$$

Evaluation Contexts $\boxed{\mathcal{C}, \mathcal{D}}$

$$\begin{array}{lcl} \mathcal{C} & ::= & \bullet \mid \mathcal{C}\ t \mid t\ \mathcal{C} \mid \lambda x{:}S.\, \mathcal{C} \mid \langle T, \mathcal{C}, c\rangle \mid \langle \mathcal{D}, t, c\rangle \mid \langle T \lhd \mathcal{D}\rangle \mid \langle \mathcal{D} \lhd S\rangle \mid \lambda x{:}\mathcal{D}.\, t \\ \mathcal{D} & ::= & x{:}\mathcal{D} \to T \mid x{:}S \to \mathcal{D} \mid \{x{:}B \mid \mathcal{C}\} \end{array}$$

performs a single evaluation step, and the relation $\leadsto^*$ is the reflexive-transitive closure of $\leadsto$. As shown in Figure 3, the rule [E-BETA] performs standard β-reduction of function applications, where we use the notation $[x \mapsto s]\, t$ to denote the capture-avoiding replacement of x by s in the term t.

The rule [E-PRIM] evaluates applications of primitive functions. This rule is defined in terms of the partial function:

$$\delta(-,-) : \mathit{Constant} \times \mathit{Term} \rightharpoonup \mathit{Term}$$

which defines the semantics of primitive functions. For example:

$$\begin{array}{rcl} \delta(\mathtt{not}, \mathtt{true}) & = & \mathtt{false} \\ \delta(+, 3) & = & +_3 \\ \delta(+_3, 4) & = & 7 \\ \delta(\mathtt{not}, 3) & = & \mathit{undefined} \\ \delta(\mathtt{if}_T, \mathtt{true}) & = & \lambda x{:}T.\, \lambda y{:}T.\, x \\ \delta(\mathtt{if}_T, \mathtt{false}) & = & \lambda x{:}T.\, \lambda y{:}T.\, y \\ \delta(\mathtt{fix}_T, t) & = & t\ \ (\mathtt{fix}_T\ t) \end{array}$$

The operational semantics of casts is a little more complicated. As described by the rule [E-CAST-FN], casting a function v of type $x{:}S_1 \to S_2$ to the type $x{:}T_1 \to T_2$ yields a new function

$$\lambda x{:}T_1.\, \langle T_2 \lhd [x \mapsto \langle S_1 \lhd T_1\rangle\ x]\, S_2\rangle\ (v\ (\langle S_1 \lhd T_1\rangle\ x))$$

This function is of the desired type $x{:}T_1 \to T_2$; it takes an argument x of type T_1, casts it to a value of type S_1, which is passed to the original function v. The result of that application has type $[x \mapsto \langle S_1 \triangleleft T_1 \rangle\ x]\, S_2$ and is then cast to the desired result type T_2. Thus, higher-order casts are performed a lazy fashion – the new casts are performed at every application of the resulting function, in a manner reminiscent of higher-order contracts[2] [Findler and Felleisen 2002]. The rules [E-Cast-Begin] and [E-Cast-End] deal with casting a basic constant c to a base refinement type $\{x\!:\!B \mid t\}$. A cast application

$$\langle \{x\!:\!B \mid t\} \triangleleft \{x\!:\!B \mid s\} \rangle\ c$$

evaluates via [E-Cast-Begin] to a cast-in-progress $\langle \{x\!:\!B \mid t\}, [x \mapsto c]\, t, c \rangle$. The instantiated predicate $[x \mapsto c]\, t$ then evaluates via the closure rule [E-Ctx]. If the predicate diverges, then the enclosing program also diverges. If the predicate evaluates to `false`, then the cast-in-progress becomes "stuck", which is called a *cast failure*. If the predicate evaluates to `true`, then c has been dynamically verified to satisfy the predicate t, and so the cast-in-progress reduces to c via [E-Cast-End]. The first component of the cast-in-progress is an annotation of the target type of the cast, used during type checking.

Note that these casts involve only predicate checks and creating checking wrappers for functions. Thus, our approach adheres to the principle of phase separation [Cardelli 1988a], in that there is no type checking of actual program syntax at run time.

2.3 The λ^H Type System

We next describe the (undecidable) λ^H type system via the collection of type judgments and rules shown in Figure 4. The judgment $E \vdash t : T$ checks that the term t has type T in environment E; the judgment $E \vdash T$ checks that T is a well-formed type in environment E; and the judgment $E \vdash S <: T$ checks that S is a subtype of T in environment E.

The rules defining these judgments are mostly straightforward. The rule [T-App] for applications differs somewhat from the rule presented in the introduction because it supports dependent function types, and because the subtyping relation is factored out into the separate subsumption rule [T-Sub]. The rule [T-Cast] ensures for a cast $\langle T \triangleleft S \rangle$ that S and T are refinements of the same simple type using the auxiliary function $\lfloor - \rfloor$ which erases dependent function types and refinements, defined as follows:

$$\begin{aligned} \lfloor \{x\!:\!B \mid t\} \rfloor &\overset{\text{def}}{=} B \\ \lfloor x\!:\!S \to T \rfloor &\overset{\text{def}}{=} \lfloor S \rfloor \to \lfloor T \rfloor \end{aligned}$$

The rule [T-Checking] ensures casts-in-progress are well-typed. For a cast-in-progress $\langle \{x\!:\!B \mid t\}, s, c \rangle$, the condition that $s \Rightarrow [x \mapsto c]\, t$ ensures that the instantiated predicate s is at least as strong as t, so if $s \leadsto^* \texttt{true}$ then $[x \mapsto c]\, t \leadsto^* \texttt{true}$ as well. We assume that variables are bound at most once in an environment.

[2] Higher-order contracts assign blame in a sophisticated fashion. Although blame assignment is compatible with our system (see *e.g.* [Gronski and Flanagan 2007]), for clarity we ignore issues of blame in this presentation.

Figure 4: Type Rules

Type rules $\boxed{E \vdash t : T}$

$$\frac{(x : T) \in E}{E \vdash x : T} \quad \text{[T-Var]}$$

$$\frac{}{E \vdash c : ty(c)} \quad \text{[T-Const]}$$

$$\frac{E \vdash S \qquad E, x : S \vdash t : T}{E \vdash (\lambda x{:}S.\, t) : (x{:}S \to T)} \quad \text{[T-Fun]}$$

$$\frac{E \vdash t_1 : (x{:}S \to T) \qquad E \vdash t_2 : S}{E \vdash t_1\ t_2 : [x \mapsto t_2]\, T} \quad \text{[T-App]}$$

$$\frac{E \vdash S \qquad E \vdash T \qquad \lfloor S \rfloor = \lfloor T \rfloor}{E \vdash \langle T \triangleleft S \rangle : S \to T} \quad \text{[T-Cast]}$$

$$\frac{\begin{array}{c} E \vdash \{x{:}B \mid t\} \qquad E \vdash c : B \qquad E \vdash s : \texttt{Bool} \\ E \vdash s \Rightarrow [x \mapsto c]\, t \end{array}}{E \vdash \langle \{x{:}B \mid t\}, s, c \rangle : \{x{:}B \mid t\}} \quad \text{[T-Checking]}$$

$$\frac{E \vdash t : S \qquad E \vdash S <: T \qquad E \vdash T}{E \vdash t : T} \quad \text{[T-Sub]}$$

Well-formed types $\boxed{E \vdash T}$

$$\frac{E \vdash S \qquad E, x : S \vdash T}{E \vdash x{:}S \to T} \quad \text{[WT-Arrow]}$$

$$\frac{E, x : B \vdash t : \texttt{Bool}}{E \vdash \{x{:}B \mid t\}} \quad \text{[WT-Base]}$$

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

$$\frac{E \vdash T_1 <: S_1 \qquad E, x : T_1 \vdash S_2 <: T_2}{E \vdash (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)} \quad \text{[S-Arrow]}$$

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

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

$$\frac{\forall \sigma.\ \text{if } \vdash \sigma : E \text{ and } \sigma(s) \leadsto^* \texttt{true} \text{ then } \sigma(t) \leadsto^* \texttt{true}}{E \vdash s \Rightarrow t} \quad \text{[Imp]}$$

Closing Substitution $\boxed{\vdash \sigma : E}$

defined in Section 5

As customary, we apply implicit α-renaming of bound variables to maintain this assumption and to ensure substitutions are capture-avoiding.

The novel aspects of this system arise from its support of subtyping between refinement types. Recall that a type $\{x:B \mid t\}$ denotes the set of constants c of type B for which $[x \mapsto c]\, t$ evaluates to `true`. Accordingly, the subtyping judgement

$$E \vdash \{x:B \mid s\} <: \{x:B \mid t\}$$

holds when s implies t under the assumption in E. As an example, the subtyping relation:

$$\emptyset \vdash \{x:\texttt{Int} \mid x > 0\} <: \{x:\texttt{Int} \mid x \geq 0\}$$

follows from the validity of the implication:

$$x : \texttt{Int} \vdash (x > 0) \Rightarrow (x \geq 0)$$

We use two auxiliary judgments to define subtyping between refinement types. The implication judgment $E \vdash s \Rightarrow t$ holds if, for all closing substitutions σ, if the term $\sigma(s)$ evaluates to `true` then $\sigma(t)$ also evaluates to `true`. Intuitively, a substitution σ (from variables to terms) is a *closing substitution* for the environment E if $\sigma(x)$ has type $E(x)$ for each $x \in \mathit{dom}(E)$. Formalizing this notion of closing substitutions in a well-defined manner is somewhat involved, so we defer this issue to Section 5.

Of course, checking implication between arbitrary predicates is undecidable, which motivates the development of the hybrid type checking algorithm in the following section.

3. HYBRID TYPE CHECKING FOR λ^H

We now describe how to perform hybrid type checking. We work in the specific context of the language λ^H, but have also demonstrated that the general approach extends to other languages with similarly expressive type systems [Gronski et al. 2006].

Hybrid type checking relies on an algorithm for conservatively approximating implication between predicates. We assume that for any conjectured implication $E \vdash s \Rightarrow t$, this algorithm returns one of three possible results, which we denote as follows:

—The judgment $E \vdash^{\surd}_{alg} s \Rightarrow t$ means the algorithm finds a proof that $E \vdash s \Rightarrow t$.

—The judgment $E \vdash^{\times}_{alg} s \Rightarrow t$ means the algorithm finds a proof that $E \not\vdash s \Rightarrow t$.

—The judgment $E \vdash^{?}_{alg} s \Rightarrow t$ means the algorithm terminates without either discovering a proof of either $E \vdash s \Rightarrow t$ or $E \not\vdash s \Rightarrow t$.

We lift this 3-valued algorithmic implication judgment $E \vdash^{a}_{alg} s \Rightarrow t$ (where $a \in \{\surd, \times, ?\}$) to a 3-valued algorithmic subtyping judgment:

$$E \vdash^{a}_{alg} S <: T$$

as shown in Figure 5. The subtyping judgment between base refinement types reduces to a corresponding implication judgment, via the rule [SA-BASE]. Subtyping between function types reduces to subtyping between corresponding contravariant

Figure 5: Cast insertion Rules

Cast insertion on terms $\boxed{E \vdash s \hookrightarrow t : T}$

$$\frac{(x : T) \in E}{E \vdash x \hookrightarrow x : T} \quad \text{[C-VAR]}$$

$$\frac{}{E \vdash c \hookrightarrow c : ty(c)} \quad \text{[C-CONST]}$$

$$\frac{E \vdash S_1 \hookrightarrow T_1 \qquad E, x : T_1 \vdash s \hookrightarrow t : T_2}{E \vdash (\lambda x{:}S_1.\, s) \hookrightarrow (\lambda x{:}T_1.\, t) : (x{:}T_1 \to T_2)} \quad \text{[C-FUN]}$$

$$\frac{E \vdash s_1 \hookrightarrow t_1 : (x{:}T_1 \to T_2) \qquad E \vdash s_2 \hookrightarrow t_2 \downarrow T_1}{E \vdash s_1\ s_2 \hookrightarrow t_1\ t_2 : [x \mapsto t_2]\, T_2} \quad \text{[C-APP]}$$

Cast insertion and checking $\boxed{E \vdash s \hookrightarrow t \downarrow T}$

$$\frac{E \vdash s \hookrightarrow t : S \qquad E \vdash^{\surd}_{alg} S <: T}{E \vdash s \hookrightarrow t \downarrow T} \quad \text{[CC-OK]}$$

$$\frac{E \vdash s \hookrightarrow t : S \qquad E \vdash^{?}_{alg} S <: T}{E \vdash s \hookrightarrow \langle T \triangleleft S \rangle\, t \downarrow T} \quad \text{[CC-CHK]}$$

Cast insertion on types $\boxed{E \vdash S \hookrightarrow T}$

$$\frac{E \vdash S_1 \hookrightarrow T_1 \qquad E, x : T_1 \vdash S_2 \hookrightarrow T_2}{E \vdash (x{:}S_1 \to S_2) \hookrightarrow (x{:}T_1 \to T_2)} \quad \text{[C-ARROW]}$$

$$\frac{E, x : B \vdash s \hookrightarrow t : \{y{:}\texttt{Bool} \mid t'\}}{E \vdash \{x{:}B \mid s\} \hookrightarrow \{x{:}B \mid t\}} \quad \text{[C-BASE]}$$

Subtyping Algorithm $\boxed{E \vdash^{a}_{alg} S <: T}$

$$\frac{E \vdash^{b}_{alg} T_1 <: S_1 \qquad E, x : T_1 \vdash^{c}_{alg} S_2 <: T_2 \qquad a = b \otimes c}{E \vdash^{a}_{alg} (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)} \quad \text{[SA-ARROW]}$$

$$\frac{E, x : B \vdash^{a}_{alg} s \Rightarrow t \qquad a \in \{\surd, \times, ?\}}{E \vdash^{a}_{alg} \{x{:}B \mid s\} <: \{x{:}B \mid t\}} \quad \text{[SA-BASE]}$$

Implication Algorithm $\boxed{E \vdash^{a}_{alg} s \Rightarrow t}$

pluggable algorithm

domain and covariant range types, via the rule [SA-ARROW]. This rule uses the following conjunction operation $\otimes$ between three-valued results:

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

If the appropriate subtyping relation holds between the domain and range components (*i.e.*, $b = c = \surd$), then the subtyping relation holds between the function types (*i.e.*, $a = \surd$). If the appropriate subtyping relation does not hold between either the domain or range components (*i.e.*, $b = \times$ or $c = \times$), then the subtyping relation does not hold between the function types (*i.e.*, $a = \times$). Otherwise, in the uncertain case, subtyping *may* hold between the function types (*i.e.*, $a = ?$). Thus, like the implication algorithm, the subtyping algorithm need not return a definite answer in all cases.

Hybrid type checking uses this subtyping algorithm to type check the source program, and to simultaneously insert dynamic casts to compensate for any indefinite answers returned by the subtyping algorithm. We characterize this process of simultaneous type checking and cast insertion via the *cast insertion judgment*:

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

Here, the environment E provides bindings for free variables, s is the original source program, t is a modified version of the original program with additional casts, and T is the inferred type for t. Since types contain terms, we extend this cast insertion process to types via the judgment $E \vdash S \hookrightarrow T$. Some of the cast insertion rules rely on the auxiliary *cast insertion and checking* judgment:

$$E \vdash s \hookrightarrow t \downarrow T$$

This judgment takes as input an environment E, a source term s, and a desired result type T, and checks that s is converted by cast insertion to a term t of this type. For simplicity, we do not allow casts in source programs; they are inserted only by the cast insertion algorithm.

The rules defining these judgments are shown in Figure 5. Most of the rules are straightforward. The rules [C-Var] and [C-Const] say that variable references and constants do not require additional casts. The rule [C-Fun] inserts casts into an abstraction $\lambda x\!:\!S_1.\,s$ by first inserting casts into the type S_1 to yield T_1 and then processing s to yield a term t of type T_2; the resulting abstraction $\lambda x\!:\!T_1.\,t$ has type $x\!:\!T_1 \rightarrow T_2$. The rule [C-App] for an application $s_1\ s_2$ processes s_1 to a term t_1 of type $x\!:\!T_1 \rightarrow T_2$ then invokes the cast insertion and checking judgement to convert s_2 into a term t_2 of the appropriate argument type T_1.

The two rules defining the cast insertion and checking judgment $E \vdash s \hookrightarrow u \downarrow T$ demonstrate the key idea of hybrid type checking. Both rules start by processing s to a term t of some type S. The crucial question is then whether this type S is a subtype of the expected type T:

—If the subtyping algorithm succeeds in proving that S is a subtype of T (*i.e.*, $E \vdash^{\surd}_{alg} S <: T$), then t is clearly of the desired type T, and so the rule [CC-Ok] returns t.

—If the subtyping algorithm can show that S is not a subtype of T (*i.e.*, $E \vdash^{\times}_{alg} S <: T$), then the program is rejected since no rule is applicable.

—Otherwise, in the uncertain case where $E \vdash^{?}_{alg} S <: T$, the rule [CC-Chk] inserts the type cast $\langle T \lhd S\rangle$ to dynamically ensure that values returned by t are actually of the desired type T.

These rules for cast insertion and checking illustrate the key benefit of hybrid type checking – specific static analysis problem instances (such as $E \vdash S <: T$) that are undecidable or computationally intractable can be avoided in a convenient manner simply by inserting appropriate dynamic checks. Of course, we should not abuse this facility, and so ideally the subtyping algorithm should yield a precise answer in most cases. However, the critical contribution of hybrid type checking is that it avoids the very strict requirement of demanding a precise answer for *all* (arbitrarily complicated) subtyping questions.

Cast insertion on types is straightforward. The rule [C-ARROW] inserts casts in the domain and codomain of a function type $x : S \to T$ and reassembles the components. The rule [C-BASE] inserts casts into the refinement s of a base type $\{x{:}B \mid s\}$, producing t (whose type should be a subtype of `Bool`), and then yielding the base refinement type $\{x{:}B \mid t\}$.

Note that checking that a type is well-formed is actually a cast insertion process that returns a well-formed type (possibly with added casts). Thus, we only perform cast insertion on types where necessary, when we encounter (possibly ill-formed) types on λ-abstractions in the source program. In particular, the cast insertion rules do not explicitly check that the environment is well-formed, since that would involve repeatedly processing all types in that environment. Instead, the rules assume that the environment is well-formed; this assumption is explicit in the correctness theorems later in the paper.

4. AN EXAMPLE

To illustrate the behavior of the cast insertion algorithm, consider a function `serializeMatrix` that serializes an n by m matrix into an array of size $n \times m$. We extend the language λ^H with two additional base types:

—`Array`, the type of one dimensional arrays containing integers.

—`Matrix`, the type of two dimensional matrices, again containing integers.

The following primitive functions return the size of an array; create a new array of the given size; and return the width and height of a matrix, respectively:

$$\begin{aligned}
\texttt{asize} &: a{:}\texttt{Array} \to \texttt{Int}\\
\texttt{newArray} &: n{:}\texttt{Int} \to \{a{:}\texttt{Array} \mid \texttt{asize}\ a = n\}\\
\texttt{matrixWidth} &: a{:}\texttt{Matrix} \to \texttt{Int}\\
\texttt{matrixHeight} &: a{:}\texttt{Matrix} \to \texttt{Int}
\end{aligned}$$

We introduce the following type abbreviations to denote arrays of size n and matrices of size n by m:

$$\begin{aligned}
\texttt{Array}_n &\overset{\text{def}}{=} \{a{:}\texttt{Array} \mid (\texttt{asize}\ a = n)\}\\
\texttt{Matrix}_{n,m} &\overset{\text{def}}{=} \{a{:}\texttt{Matrix} \mid \begin{pmatrix} \texttt{matrixWidth}\ \ a = n \\ \wedge\ \texttt{matrixHeight}\ a = m \end{pmatrix}\}
\end{aligned}$$

The shorthand t `as` T ensures that the term t has type T by passing t as an argument to the identity function of type $T \to T$:

$$t\ \texttt{as}\ T \overset{\text{def}}{=} (\lambda x{:}T.\, x)\ t$$

We now define the function `serializeMatrix` as:

$$\left(\begin{array}{l}\lambda n\!:\!\texttt{Int}.\ \lambda m\!:\!\texttt{Int}.\ \lambda a\!:\!\texttt{Matrix}_{n,m}.\\ \quad \texttt{let}\ r\ =\ \texttt{newArray}\ e\ \texttt{in}\ \ldots\ ;\ r\end{array}\right)\ \texttt{as}\ T$$

The elided term ... initializes the new array r with the contents of the matrix a, and we will consider several possibilities for the size expression e. The type T is the specification of `serializeMatrix`:

$$T\ \overset{\text{def}}{=}\ (n\!:\!\texttt{Int} \rightarrow\ m\!:\!\texttt{Int} \rightarrow\ \texttt{Matrix}_{n,m} \rightarrow\ \texttt{Array}_{n\times m})$$

For this declaration to type check, the inferred type $\texttt{Array}_e$ of the function's body must be a subtype of the declared return type:

$$n : \texttt{Int},\ m : \texttt{Int}\ \vdash \texttt{Array}_e <: \texttt{Array}_{n\times m}$$

Checking this subtype relation reduces to checking the implication:

$$\begin{array}{rl} n : \texttt{Int},\ m : \texttt{Int},\ r : \texttt{Array}\ \vdash & \ \ (\texttt{asize}\ r = e)\\ & \Rightarrow\ (\texttt{asize}\ r = (n \times m))\end{array}$$

which in turn reduces to checking the equality:

$$\forall n, m \in \texttt{Int}.\ \ e = n \times m$$

The implication checking algorithm might use an automatic theorem prover (*e.g.*, [Detlefs et al. 2005; Blei et al. 2000]) to verify or refute such conjectured equalities.

We now consider three possibilities for the expression e.

(1) If e is the expression $n \times m$, the equality is trivially true, and no additional casts are inserted (even when using a rather weak theorem prover).

(2) If e is $m \times n$ (*i.e.*, the order of the multiplicands is reversed), and the underlying theorem prover can verify

$$\forall n, m \in \texttt{Int}.\ \ m \times n = n \times m$$

then again no casts are necessary. Note that a theorem prover that is not complete for arbitrary multiplication might still have a specific axiom about the commutativity of multiplication.

If the theorem prover is too limited to verify this equality, the hybrid type checker will still accept this program. However, to compensate for the limitations of the theorem prover, the hybrid type checker will insert a redundant cast, yielding the function (where we have elided the source type of the cast):

$$\left(\langle T \triangleleft \ldots\rangle\ \left(\begin{array}{l}\lambda n\!:\!\texttt{Int}.\ \lambda m\!:\!\texttt{Int}.\ \lambda a\!:\!\texttt{Matrix}_{n,m}.\\ \quad \texttt{let}\ r\ =\ \texttt{newArray}\ e\ \texttt{in}\ \ldots\ ;\ r\end{array}\right)\right)\ \texttt{as}\ T$$

This term can be optimized, via [E-Beta] and [E-Cast-Fn] steps and via removal of clearly redundant $\langle\texttt{Int} \triangleleft \texttt{Int}\rangle$ casts, to:

$$\begin{array}{l}\lambda n\!:\!\texttt{Int}.\ \lambda m\!:\!\texttt{Int}.\ \lambda a\!:\!\texttt{Matrix}_{n,m}.\\ \quad \texttt{let}\ r\ =\ \texttt{newArray}\ (m \times n)\ \texttt{in}\\ \qquad\quad \ldots\ ;\\ \qquad\quad \langle\texttt{Array}_{n\times m} \triangleleft \texttt{Array}_{m\times n}\rangle\ r\end{array}$$

The remaining cast checks that the result value r is of the declared return type $\texttt{Array}_{n\times m}$, which reduces to dynamically checking that the predicate:

$$\texttt{asize}\ r \quad = \quad n \times m$$

evaluates to `true`, which it does.

(3) Finally, if e is erroneously $m \times m$, the function is ill-typed. By performing random or directed [Godefroid et al. 2005] testing of several values for n and m until it finds a counterexample, the theorem prover might reasonably refute the conjectured equality:

$$\forall n, m \in \texttt{Int}.\ \ m \times m = n \times m$$

In this case, the hybrid type checker reports a static type error.

Conversely, if the theorem prover is too limited to refute the conjectured equality, then the hybrid type checker will produce (after optimization) the program:

$$\begin{array}{l} \lambda n\!:\!\texttt{Int}.\ \lambda m\!:\!\texttt{Int}.\ \lambda a\!:\!\texttt{Matrix}_{n,m}. \\ \quad \texttt{let}\ r\ =\ \texttt{newArray}\ (m \times m)\ \texttt{in} \\ \qquad \ldots\ ; \\ \qquad \langle \texttt{Array}_{n\times m} \triangleleft \texttt{Array}_{m\times m} \rangle\ r \end{array}$$

If this function is ever called with arguments for which $m \times m \neq n \times m$, then the cast will detect the type error.

Note that prior work on practical dependent types [Xi and Pfenning 1999] could not handle these cases, since the type T uses non-linear arithmetic expressions. In contrast, case 2 of this example demonstrates that even fairly partial techniques for reasoning about complex specifications (*e.g.*, commutativity of multiplication, random testing of equalities) can facilitate static detection of defects. Furthermore, even though catching errors at compile time is ideal, catching errors at run time (as in case 3) is still an improvement over not detecting these errors at all, and getting subsequent crashes or incorrect results.

5. CLOSING SUBSTITUTIONS

We now define the closing substitution relation, deferred from Section 2.3, which lies at the heart of our type system. One approach is to define closing substitutions in terms of the typing judgement, as follows [Flanagan 2006]:

$$\frac{\forall x \in dom(E),\ \ \vdash \sigma(x)\ :\ E(x)}{\vdash \sigma\ :\ E}$$

But this approach leads to a cyclic definition between the typing, subtyping, implication, and closing substitution judgements. Moreover, the implication rule [Imp] from Figure 4 refers to the closing substitution relation in a negative position. Thus, standard monotonicity arguments are not sufficient to show that the resulting collection of mutually-recursive type rules are actually well-defined, and it is not obvious if there are interesting type systems that satisfy these rules.

An alternative approach to defining implication is to axiomatize its logic [Denney 1998; Ou et al. 2004; Gronski et al. 2006], but the underlying problem remains, in that it is still not obvious if there are interesting implication relations (and hence type systems) that satisfy these axioms.

To provide a solid foundation for contract types, we "bootstrap" the type system by using a denotational interpretation of contract types. A contract type has a natural interpretation as a dynamic contract on top of the simply-typed lambda calculus, and we formalize this connection as follows. First, we extend the $\lfloor - \rfloor$ function defined in Section 2.3 to terms. For each term t, erasing all refinements and eliminating casts yields a term $\lfloor t \rfloor$ of the simply-typed lambda calculus.

$$\begin{array}{rcl}
\lfloor c \rfloor & \stackrel{\text{def}}{=} & c \\
\lfloor x \rfloor & \stackrel{\text{def}}{=} & x \\
\lfloor \lambda x\!:\!S.\, t \rfloor & \stackrel{\text{def}}{=} & \lambda x\!:\!\lfloor S \rfloor.\, \lfloor t \rfloor \\
\lfloor t_1\ t_2 \rfloor & \stackrel{\text{def}}{=} & \lfloor t_1 \rfloor\ \lfloor t_2 \rfloor \\
\lfloor \langle T \triangleleft S \rangle \rfloor & \stackrel{\text{def}}{=} & \lambda x\!:\!\lfloor S \rfloor.\, x \\
\lfloor \langle T, t, c \rangle \rfloor & \stackrel{\text{def}}{=} & c
\end{array}$$

Next, we give each type T an interpretation $[\![T]\!]$ (defined by induction on $\lfloor T \rfloor$) that is the set of terms t such that $\lfloor t \rfloor$ is of type $\lfloor T \rfloor$ and t obeys the contractual aspect of T.

$$\begin{array}{rcl}
[\![\{x\!:\!B \mid s\}]\!] & \stackrel{\text{def}}{=} & \{t \mid\ \vdash_{\text{STLC}} \lfloor t \rfloor : B\ \wedge\ (t \leadsto^* c \text{ implies } [x \mapsto c]\, s \leadsto^* \texttt{true})\} \\
[\![x\!:\!S \to T]\!] & \stackrel{\text{def}}{=} & \{t \mid\ \vdash_{\text{STLC}} \lfloor t \rfloor : \lfloor S \rfloor \to \lfloor T \rfloor\ \wedge\ \forall s \in [\![S]\!],\ t\ s \in [\![[x \mapsto s]\, T]\!]\}
\end{array}$$

Here, we use $\vdash_{\text{STLC}}$ to denote the standard typing relation for the simply-typed lambda calculus. A refined base type $\{x : B \mid s\}$ is interpreted as the set of closed terms t of type B for which the predicate s holds, in the sense that whenever $t \leadsto^* c$ for some constant c, then $[x \mapsto c]\, s \leadsto^* \texttt{true}$. Note that we cannot insist on the simpler requirement that $[x \mapsto t]\, s \leadsto^* \texttt{true}$ because that would forbid assigning types to divergent terms. For example, any type T is populated by at least the divergent term $\texttt{fix}_T\ (\lambda x\!:\!T.\, x)$. In other terminology, contract types specify *partial* correctness, not *total* correctness.

A dependent function type $x\!:\!S \to T$ is interpreted as the set of closed terms of simple type $\lfloor S \rfloor \to \lfloor T \rfloor$ that give output in $[\![[x \mapsto s]\, T]\!]$ whenever their input s is in $[\![S]\!]$.

Based on this notion of semantic typing, we are now in a position to define closing substitutions while avoiding circularity problems. The following rule [Subst] judges that a substitution $\sigma : Var \to Term$ is a *closing substitition* for environment E if for all bindings $(x : T) \in E$ we have $\sigma(x) \in \sigma(T)$; in this case we write $\vdash \sigma : E$.

$$\frac{\forall x \in dom(E),\ \sigma(x) \in [\![\sigma(E(x))]\!]}{\vdash \sigma : E}\,[\text{Subst}]$$

6. CORRECTNESS

Having completed the formal definition of our type system, we now proceed to study the correctness properties that are guaranteed by hybrid type checking. We begin with the type system, which provides the specification for our hybrid cast insertion algorithm.

6.1 Correctness of the Type System

We prove soundness of our type system by the syntactic method via *progress* and *preservation* (or subject reduction) theorems [Wright and Felleisen 1994]. The preservation theorem includes a requirement that the environment E is well-formed ($\vdash E$), a notion that is defined in Figure 6. Note that the type rules do not refer to this judgment directly in order to yield a closer correspondence with the cast insertion rules. For the remainder of this section, we elide this condition and assume all environments are well-formed.

As usual, a term is considered to be in *normal form* if it does not reduce to any subsequent term, and a *value* v is either a λ-abstraction, a type cast, or a constant. We assume that the function *ty* maps each constant to an appropriate type, in the following sense:

Assumption 1 (Types of Constants). *For each $c \in$ Constant:*

(*1*) *The type of c is closed and well-formed,* i.e. $\emptyset \vdash ty(c)$.

(*2*) *If c is a primitive function then it cannot get stuck and its behavior is compatible with its type,* i.e. *if* $\emptyset \vdash c\ v : T$ *then* $\delta(c, v)$ *is defined and* $\emptyset \vdash \delta(c, v) : T$

(*3*) *If c is a basic constant then it is a member of its type, which is a singleton type,* i.e.*if* $ty(c) = \{x : B \mid t\}$ *then* $[x \mapsto c]\, t \leadsto^* \texttt{true}$ *and* $\forall c' \neq c.\ [x \mapsto c']\, t \not\leadsto^* \texttt{true}$.

The central lemma for proving preservation is the so-called *substitution lemma*, which states that a variable of a certain type may be soundly replaced by any term of the same type. Because we define closing substitutions semantically, our proof of the substitution lemma is nonstandard and connects the semantic relation with our formal type system.

First, we define the *semantic subtyping* relation $E \vdash S \subseteq T$ and the *semantic typing* relation $E \vdash t \in T$ induced by our semantic notion of types, following [Frisch et al. 2002]:

$$\begin{array}{rcl} E \vdash S \subseteq T & \overset{\text{def}}{=} & \forall \sigma,\ \vdash \sigma : E \text{ implies } [\![\sigma(S)]\!] \subseteq [\![\sigma(T)]\!] \\ E \vdash t \in T & \overset{\text{def}}{=} & \forall \sigma,\ \vdash \sigma : E \text{ implies } \sigma(t) \in [\![\sigma(T)]\!] \end{array}$$

Our formal subtype system is sound with respect to this model:

Lemma 2.

(*1*) *If* $E \vdash S <: T$ *then* $E \vdash S \subseteq T$

(*2*) *If* $E \vdash t : T$ *then* $E \vdash t \in T$

Proof.

(1) By induction on the derivation of $E \vdash S <: T$. If S and T are refined base types, then semantic and formal subtyping have identical definitions. If S and T are function types, then the result follows by induction.

(2) By induction on the derivation of $E \vdash t : T$. In the case for [T-Sub] we invoke part (1).

Figure 6: Well-formed Environments

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

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

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

□

The formal substitution lemma now follows from the connection between the formal and semantic relations.

LEMMA 3 (SUBSTITUTION). *Suppose $E \vdash s : S$,*

(1) If $E, x : S, F \vdash T <: U$ then $E, [x \mapsto s]\, F \vdash [x \mapsto s]\, T <: [x \mapsto s]\, U$
(2) If $E, x : S, F \vdash t : T$ then $E, [x \mapsto s]\, F \vdash [x \mapsto s]\, t : [x \mapsto s]\, T$
(3) If $E, x : S, F \vdash T$ then $E, [x \mapsto s]\, F \vdash [x \mapsto s]\, T$

PROOF.
(1) By induction on the derivation of $E, x : S, F \vdash T <: U$. In the case of [S-BASE], $T = \{y : B \mid t_1\}$ and $U = \{y : B \mid t_2\}$ and we have $E, x : S, F, y : B \vdash t_1 \Rightarrow t_2$. By Lemma 2(2), $E \vdash s \in S$ so $E, [x \mapsto s]\, F, y : B \vdash [x \mapsto s]\, t_1 \Rightarrow [x \mapsto s]\, t_2$ is immediate.
(2) and (3) By mutual induction on the derivations of $E, x : S, F \vdash t : T$ and $E, x : S, F \vdash T$, invoking part (1) in the case of [T-SUB]. □

THEOREM 4 (PRESERVATION). *If $E \vdash s : T$ and $s \leadsto t$ then $E \vdash t : T$*

PROOF. By induction on the typing derivation $E \vdash s : T$, invoking Lemma 3 when evaluation proceeds via [E-BETA]. □

The progress property of our type system includes the caveat that type casts may fail. A failed cast is one that casts a constant to a refinement type with an incompatible predicate.

THEOREM 5 (PROGRESS). *Every well-typed, closed normal form is either a value or contains a failed cast.*

PROOF. By induction of the derivation showing that the normal form is well-typed. □

6.2 Type Correctness of Cast Insertion

Since hybrid type checking relies on necessarily incomplete algorithms for subtyping and implication, we next investigate what correctness properties are guaranteed by this cast insertion process.

We assume the 3-valued algorithm for checking implication between boolean terms is sound in the following sense:

Assumption 6 (Soundness of $E \vdash^{a}_{alg} s \Rightarrow t$). *Suppose* $\vdash E$.

(1) *If* $E \vdash^{\checkmark}_{alg} s \Rightarrow t$ *then* $E \vdash s \Rightarrow t$.

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

Note that this algorithm does not need to be complete (indeed, an extremely naive algorithm could simply return $E \vdash^{?}_{alg} s \Rightarrow t$ in all cases). A consequence of the soundness of the implication algorithm is that the algorithmic subtyping judgment $E \vdash_{alg} S <: T$ is also sound.

Lemma 7 (Soundness of $E \vdash^{a}_{alg} S <: T$). *Suppose* $\vdash E$.

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

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

Proof. By induction on derivations using Assumption 6. $\square$

Because algorithmic subtyping is sound, the hybrid cast insertion algorithm generates only well-typed programs:

Theorem 8 (Compilation Soundness). *Suppose* $\vdash E$.

(1) *If* $E \vdash t \hookrightarrow t' : T$ *then* $E \vdash t' : T$.

(2) *If* $E \vdash t \hookrightarrow t' \downarrow T$ *and* $E \vdash T$ *then* $E \vdash t' : T$.

(3) *If* $E \vdash T \hookrightarrow T'$ *then* $E \vdash T'$.

Proof. By induction on cast insertion derivations. $\square$

Since the generated code is well-typed, standard type-directed optimization techniques are applicable. Furthermore, the generated code includes all the type specifications present in the original program, and so by the Preservation Theorem these specifications will never be violated at run time. Any attempt to violate a specification is detected via a combination of static checking (where possible) and dynamic checking (when necessary).

6.3 Behavioral Correctness of Cast Insertion

We now prove that cast insertion does not change the behavior of well-typed programs, in the sense that the original and compiled programs are equivalent in any well-typed context. Our proof is based on the technique of *logical relations* [Statman 1985]. The logical relation we use is extensional equivalence of well-typed terms under reduction, written $E \vdash s \sim t : T$ and defined in Figure 7. (As we shall see below, extensional equivalence implies the traditional notion of contextual equivalence.)

Two terms t_1 and t_2 of base type are (extensionally) equivalent if, for any closing substitution σ, whenever $\sigma(t_1)$ evaluates to a constant c so does $\sigma(t_2)$, and vice versa. Two functions t_1 and t_2 of type $x : S \to T$ are equivalent if they yield equivalent output when given equivalent arguments s_1 and s_2.

A question that immediately arises is whether these two applications $t_1\ s_1$ and $t_2\ s_2$ should have type $[x \mapsto s_1]\,T$ or $[x \mapsto s_2]\,T$. In some sense it does not matter, since s_1 and s_2 are extensionally equivalent by assumption. However, it significantly

Figure 7: Extensional Equivalence Under Reduction

$\boxed{E \vdash t_1 \sim t_2 : T}$ (defined by induction on $\lfloor T \rfloor$)

$$\begin{array}{lcl}
E \vdash t_1 \sim t_2 : \{x{:}B \mid p\} & \Leftrightarrow & E \vdash t_1 \in \{x{:}B \mid p\} \\
 & & E \vdash t_2 \in \{x{:}B \mid p\} \\
 & & \forall \sigma \text{ such that } \vdash \sigma : E,\ (\sigma(t_1) \leadsto^* c) \Leftrightarrow (\sigma(t_2) \leadsto^* c) \\
 & & \\
E \vdash t_1 \sim t_2 : (x{:}S \to T) & \Leftrightarrow & \forall s_1, s_2 \text{ such that } E \vdash s_1 \sim s_2 : S, \\
 & & E \vdash t_1\ s_1 \sim t_2\ s_2 : [x \mapsto s_1]\,T \curlywedge [x \mapsto s_2]\,T
\end{array}$$

$\boxed{S \curlywedge T}$

$$\begin{array}{lcl}
\{x{:}B \mid p\} \curlywedge \{x{:}B \mid q\} & = & \{x{:}B \mid p \wedge q\} \\
 & & \\
(x{:}S_1 \to S_2) \curlywedge (x{:}T_1 \to T_2) & = & x{:}(S_1 \curlywedge T_1) \to (S_2 \curlywedge T_2)
\end{array}$$

simplifies our proof structure to require that $t_1\ s_1$ and $t_2\ s_2$ are extensionally equivalent at type

$$[x \mapsto s_1]\,T \curlywedge [x \mapsto s_2]\,T$$

where we combine both types via the *wedge product* ($\curlywedge$) defined in Figure 7. For base types, wedge product is type intersection, *i.e.*, conjunction of refinement predicates. For function types, the wedge product is covariant in both the function domain and the range. Though this covariance may seem surprising, we we only ever combine equivalent types with the wedge product, so co- or contravariance is a formality: If the wedge product were made more "intuitively" contravariant the [T-App] case of Theorem 12 would not be possible to prove as stated.

To work fluidly with this wedge product, we note that it is a commutative and associative operator with respect to the equivalence closure of subtyping, and it distributes through subtyping.

Lemma 9 (Wedge Product Subtyping). *Assuming the underlying shapes of all mentioned types are equal, and each type is well-formed in the environment,*

(1) $E \vdash S \curlywedge T <: T \curlywedge S$

(2) $E \vdash S \curlywedge (T \curlywedge U) <: (S \curlywedge T) \curlywedge U$

(3) If $E \vdash S_1 <: T_1$ *and* $E \vdash S_2 <: T_2$ *then* $E \vdash S_1 \curlywedge S_2 <: T_1 \curlywedge T_2$

Proof. By induction on the underlying shape of the types. □

We assume that primitive functions do not violate extensionality, and so each primitive function c is extensionally equivalent to itself, *i.e.*, $\emptyset \vdash c \sim c : ty(c)$. Next, equivalence is preserved under subtyping, allowing free manipulation of $\curlywedge$ in the type assigned to an equivalence.

Lemma 10 Extensional equivalence under subtyping. *If* $E \vdash s \sim t : S$ *and* $E \vdash S <: T$ *then* $E \vdash s \sim t : T$

Proof. By induction on $\lfloor S \rfloor$ (which equals $\lfloor T \rfloor$)

Case $S = \{x\!:\!B \mid p\}$: In this case, s and t are also semantically members of T, by definition of subtyping between base types, and have the same reducts.

Case $S = x\!:\!S_1 \to S_2$ and $T = x\!:\!T_1 \to T_2$.
By inversion,

$$E \vdash T_1 <: S_1 \quad \text{and} \quad E, x : T_1 \vdash S_2 <: T_2$$

Fix a and b such that $E \vdash a \sim b : T_1$. By induction, $E \vdash a \sim b : S_1$.
By definition,

$$E \vdash s\ a \sim t\ b : [x \mapsto a]\, S_2 \curlywedge [x \mapsto b]\, S_2$$

By the Substitution Lemma 3,

$$E \vdash [x \mapsto a]\, S_2 <: [x \mapsto a]\, T_2 \quad \text{and} \quad E \vdash [x \mapsto b]\, S_2 <: [x \mapsto b]\, T_2$$

By Lemma 9(3),

$$E \vdash [x \mapsto a]\, S_2 \curlywedge [x \mapsto b]\, S_2 <: [x \mapsto a]\, T_2 \curlywedge [x \mapsto b]\, T_2$$

By induction,

$$E \vdash s\ a \sim t\ b : [x \mapsto a]\, T_2 \curlywedge [x \mapsto b]\, T_2$$

Hence $E \vdash s \sim t : (x\!:\!T_1 \to T_2)$ $\square$

Next, we show that extensional equivalence of reducts implies extensional equivalence of the original terms.

Lemma 11 Extensional equivalence under reduction. *If* $E \vdash s \sim t : T$ *and* $s' \leadsto^* s$ *and* $t' \leadsto^* t$ *then* $E \vdash s' \sim t' : T$

Proof. By induction on $\lfloor T \rfloor$.
Case $T = \{x\!:\!B \mid p\}$: If $s \leadsto^* c$ then $s' \leadsto^* s \leadsto^* c$; likewise for t and t'.

Case $T = x\!:\!T_1 \to T_2$: Fix a, b such that

$$E \vdash a \sim b : T_1.$$

By definition

$$E \vdash s\ a \sim t\ b : [x \mapsto a]\, T_2 \curlywedge [x \mapsto b]\, T_2$$

By applying an evaluation context,

$$s'\ a \leadsto^* s\ a \quad \text{and} \quad t'\ b \leadsto^* t\ b$$

By induction,

$$E \vdash s'\ a \sim t'\ b : [x \mapsto a]\, T_2 \curlywedge [x \mapsto b]\, T_2$$

Hence, $E \vdash s' \sim t' : (x\!:\!T_1 \to T_2)$. $\square$

To prove the fundamental soundness theorem for extensional equivalence, we extend the notion extensional equivalence to closing substitutions.

$$\vdash \sigma \sim \gamma : E \quad \overset{\text{def}}{=} \quad \forall x \in \mathit{dom}(E),\ \emptyset \vdash \sigma(x) \sim \gamma(x) : (\sigma \curlywedge \gamma)(E(x))$$

For convenience, we overload the wedge product operator for closing substitutions, so $(\sigma \curlywedge \gamma)$ maps a type T to the wedge product of all possible choices of which variables to use from σ and which from γ.

$$(\sigma \curlywedge \gamma)(T) \overset{\text{def}}{=} \bigcurlywedge_{A \in \mathcal{P}(dom(\sigma))} (\sigma|_A \circ \gamma|_{dom(\sigma)\setminus A})(T)$$

THEOREM 12 SOUNDNESS OF EXTENSIONAL EQUIVALENCE. *If $E \vdash t : T$ then for any closing substitutions σ and γ such that $\vdash \sigma \sim \gamma : E$, we have*

$$\emptyset \vdash \sigma(t) \sim \gamma(t) : (\sigma \curlywedge \gamma)(T)$$

PROOF. By induction on the height of the derivation of $E \vdash t : T$.

Case [T-VAR]. By inversion, $(x : T) \in E$, so by definition of equivalent substitutions $\emptyset \vdash \sigma(x) \sim \gamma(x) : (\sigma \curlywedge \gamma)(T)$

Case [T-PRIM] By the above assumption on constants.

Case [T-APP] By inversion,

$$\begin{array}{ll} t = s_1\ s_2 & T = [x \mapsto s_2]\, S_2 \\ E \vdash s_1 : (x{:}S_1 \to S_2) & E \vdash s_2 : S_1 \end{array}$$

Fix σ and γ. By induction,

$$\begin{array}{l} \emptyset \vdash \sigma(s_1) \sim \gamma(s_1) : (\sigma \curlywedge \gamma)(x{:}S_1 \to S_2) \\ \emptyset \vdash \sigma(s_2) \sim \gamma(s_2) : (\sigma \curlywedge \gamma)(S_1) \end{array}$$

By definition,

$$\begin{array}{rcl} \emptyset \vdash \sigma(s_1)\ \sigma(s_2) \sim \gamma(s_1)\ \gamma(s_2) & : & [x \mapsto \sigma(s_2)]\,(\sigma \curlywedge \gamma)(S_2) \\ & \curlywedge & [x \mapsto \gamma(s_2)]\,(\sigma \curlywedge \gamma)(S_2) \end{array}$$

which simplifies to

$$\emptyset \vdash \sigma(s_1\ s_2) \sim \gamma(s_1\ s_2) : (\sigma \curlywedge \gamma)([x \mapsto s_2]\, S_2)$$

Case [T-LAM] By inversion,

$$t = \lambda x{:}S_1.\, s \qquad T = x{:}S_1 \to S_2 \qquad E, x : S_1 \vdash s : S_2$$

Fix σ and γ, then fix a and b such that $\emptyset \vdash a \sim b : (\sigma \curlywedge \gamma)(S_1)$. Then we have

$$\vdash (\sigma \circ [x \mapsto a]) \sim (\gamma \circ [x \mapsto b]) : (E, x : S_1)$$

By induction,

$$\emptyset \vdash (\sigma \circ [x \mapsto a])\ s \sim (\gamma \circ [x \mapsto b])\ s : ((\sigma \circ [x \mapsto a]) \curlywedge (\gamma \circ [x \mapsto b]))(S_2)$$

By Lemma 11, we perform β-expansion and rearrange some substitutions to yield

$$\begin{array}{rcl} \emptyset \vdash \sigma(\lambda x{:}S_1.\, s)\ a \sim \gamma(\lambda x{:}S_1.\, s)\ b & : & [x \mapsto a]\,(\sigma \curlywedge \gamma)(S_2) \\ & \curlywedge & [x \mapsto b]\,(\sigma \curlywedge \gamma)(S_2) \end{array}$$

which simplifies to, by definition,

$$\emptyset \vdash \sigma(\lambda x\!:\!S_1.\, s) \;\sim\; \gamma(\lambda x\!:\!S_1.\, s) : (\sigma \curlywedge \gamma)(x\!:\!S_1 \to S_2)$$

Case [T-CAST] There are two cases to consider.

SubCase $t = \langle \{x\!:\!B \mid q\} \triangleleft \{x\!:\!B \mid p\}\rangle \quad T = \{x\!:\!B \mid p\} \to \{x\!:\!B \mid q\}$

Fix a and b such that $\emptyset \vdash a \;\sim\; b : (\sigma \curlywedge \gamma)(\{x\!:\!B \mid p\})$.

By definition of equivalence at base type, a and b either diverge or both terminate at a constant c. Since base type casts are strict in their argument, the divergent case is trivial, so assume

$$a \leadsto^* c \qquad\qquad b \leadsto^* c$$

Consider the evaluation of $\sigma([x \mapsto c]\, q)$ and $\gamma([x \mapsto c]\, q)$. The typing of q is a subderivation, and $\vdash \sigma \circ [x \mapsto c] \;\sim\; \gamma \circ [x \mapsto c] : (E, x : B)$ so by induction

$$\emptyset \vdash \sigma([x \mapsto c]\, q) \;\sim\; \gamma([x \mapsto c]\, q) : \texttt{Bool}$$

By definition of logical equivalence at base type, these two boolean terms evaluate to the same truth-value or diverge together. So either both casts fail (divergence) or both succeed, evaluating to c, proving this case of the lemma.

SubCase $t = \langle x\!:\!T_1 \to T_2 \triangleleft x\!:\!S_1 \to S_2\rangle \quad T = (x\!:\!S_1 \to S_2) \to (x\!:\!T_1 \to T_2)$

Fix σ and γ and also f and g such that $\emptyset \vdash f \;\sim\; g : (\sigma \curlywedge \gamma)(x\!:\!S_1 \to S_2)$.

We need to show that

$$\begin{array}{ll} \emptyset \vdash & \sigma(\langle x\!:\!T_1 \to T_2 \triangleleft x\!:\!S_1 \to S_2\rangle)\; g \\ & \sim\; \gamma(\langle x\!:\!T_1 \to T_2 \triangleleft x\!:\!S_1 \to S_2\rangle)\; f \;:\; (\sigma \curlywedge \gamma)(x\!:\!T_1 \to T_2) \end{array}$$

Now fix a and b such that $\emptyset \vdash a \;\sim\; b : (\sigma \curlywedge \gamma)(T_1)$ and by Lemma 11 it suffices to show equivalence of the following reducts of the casts:

$$\begin{array}{lll} \emptyset \vdash & \sigma(\langle T_2 \triangleleft [x \mapsto \langle S_1 \triangleleft T_1\rangle\; a]\, S_2\rangle)\; (f\; (\sigma(\langle S_1 \triangleleft T_1\rangle)\; a) & \\ \sim & \gamma(\langle T_2 \triangleleft [x \mapsto \langle S_1 \triangleleft T_1\rangle\; b]\, S_2\rangle)\; (g\; (\gamma(\langle S_1 \triangleleft T_1\rangle)\; b) & : \; [x \mapsto a]\, (\sigma \curlywedge \gamma)(T_2) \\ & & \curlywedge \; [x \mapsto b]\, (\sigma \curlywedge \gamma)(T_2) \end{array}$$

By induction (recalling that induction is over the *height* of the derivations, since these casts were not initially subterms),

$$\emptyset \vdash \sigma(\langle S_1 \triangleleft T_1\rangle) \;\sim\; \gamma(\langle S_1 \triangleleft T_1\rangle) : (\sigma \curlywedge \gamma)(T_1 \to S_1)$$

so (noting that the function type is non-dependent)

$$\emptyset \vdash \sigma(\langle S_1 \triangleleft T_1\rangle)\; a \;\sim\; \gamma(\langle S_1 \triangleleft T_1\rangle)\; b : (\sigma \curlywedge \gamma)(S_1)$$

continuing, by combining substitutions

$$\begin{array}{l} \emptyset \vdash f\; (\sigma(\langle S_1 \triangleleft T_1\rangle)\; a) \;\sim\; g\; (\gamma(\langle S_1 \triangleleft T_1\rangle)\; b) \\ \quad : ((\sigma \circ [x \mapsto \langle S_1 \triangleleft T_1\rangle\; a]) \curlywedge (\gamma \circ [x \mapsto \langle S_1 \triangleleft T_1\rangle\; b]))\; (S_2) \end{array}$$

Also by induction,

$$\begin{array}{rl} \emptyset \vdash & (\sigma \circ [x \mapsto a])(\langle T_2 \lhd [x \mapsto \langle S_1 \lhd T_1 \rangle\ x]\, S_2 \rangle) \\ \sim & (\gamma \circ [x \mapsto b])(\langle T_2 \lhd [x \mapsto \langle S_1 \lhd T_1 \rangle\ x]\, S_2 \rangle) \\ : & ((\sigma \circ [x \mapsto a]) \curlywedge (\gamma \circ [x \mapsto b]))\ ([x \mapsto \langle S_1 \lhd T_1 \rangle\ x]\, S_2 \to T_2) \end{array}$$

so noting these equivalences:

$$\begin{array}{rl} & ((\sigma \circ [x \mapsto a]) \curlywedge (\gamma \circ [x \mapsto b]))\ \ ([x \mapsto \langle S_1 \lhd T_1 \rangle\ x]\, S_2) \\ \equiv & ((\sigma \circ [x \mapsto \langle S_1 \lhd T_1 \rangle\ a]) \curlywedge (\gamma \circ [x \mapsto \langle S_1 \lhd T_1 \rangle\ b]))\ (S_2) \\ \\ & ((\sigma \circ [x \mapsto a]) \curlywedge (\gamma \circ [x \mapsto b]))\ (T_2) \\ \equiv & [x \mapsto a]\, (\sigma \curlywedge \gamma)(T_2)\ \curlywedge\ [x \mapsto b]\, (\sigma \curlywedge \gamma)(T_2) \end{array}$$

we gain exactly the conclusion desired.

Case [T-SUB] By inversion,

$$E \vdash t : S \qquad E \vdash S <: T$$

Fix σ and γ, and by induction,

$$\emptyset \vdash \sigma(t) \sim \gamma(t) : (\sigma \curlywedge \gamma)(S)$$

Applying the substitution lemma and distributing $\curlywedge$ through the subtyping,

$$\emptyset \vdash (\sigma \curlywedge \gamma)(S) <: (\sigma \curlywedge \gamma)(T)$$

Thus by Lemma 10 we conclude with

$$\emptyset \vdash \sigma(s) \sim \gamma(s) : (\sigma \curlywedge \gamma)(T)$$

□

As a corollary, we have contextual equivalence of related terms.

COROLLARY 13. *If $E \vdash s \sim t : T$ then for any context $\mathcal{C}$ such that $\emptyset \vdash \mathcal{C}[s] : B$ and $\emptyset \vdash \mathcal{C}[t] : B$, we have that $\mathcal{C}[s] \leadsto^* c \Leftrightarrow \mathcal{C}[t] \leadsto^* c$.*

PROOF. Apply soundness of extensional equivalence to the typing $x : T \vdash \mathcal{C}[x] : B$ with the related substitutions $\vdash [x \mapsto s] \sim [x \mapsto t] : (x : T)$. □

To complete the proof that insertion of upcasts never changes the behavior of a program, we prove that *upcast* from a subtype S to a supertype T is extensionally (and hence contextually) equivalent to the appropriately-typed identity function.

THEOREM 14. *If $E \vdash S <: T$ then $E \vdash \langle T \lhd S \rangle \sim (\lambda x{:}S.\, x) : (S \to T)$*

PROOF. By induction on the height of S (which equals the height of T since they must have the same shape).

Suppose $S = \{x{:}B \mid t_1\}$ and $T = \{x{:}B \mid t_2\}$. Then to test equivalence of $\langle T \lhd S \rangle$ and $(\lambda x{:}S.\, x)$, we apply them to two terms t_3 and t_4 such that $E \vdash t_3 \sim t_4 : S$. For any closing substitution σ, we are guaranteed that $\sigma(t_3) \leadsto^* c$ if and only if $\sigma(t_4) \leadsto^* c$, so it suffices to consider applying the cast to constants. By the semantic definition of implication and the assumption that constants are assigned appropriate types, $\sigma([x \mapsto c]\, t_2) \leadsto^*$ `true` so the cast succeeds, behaving as the identity function on c.

Now suppose $S = x : S_1 \to S_2$ and $T = x : T_1 \to T_2$. Applying the cast $\langle T \triangleleft S \rangle$ to a function s of type S, yields $\lambda x : T_1.\ \langle T_2 \triangleleft [x \mapsto \langle S_1 \triangleleft T_1 \rangle\ x]\ S_2 \rangle\ (s\ (\langle S_1 \triangleleft T_1 \rangle\ x))$. For an argument t of type T_1, we have by induction that $\langle S_1 \triangleleft T_1 \rangle\ t$ is equivalent to t and $\langle [x \mapsto t]\ T_2 \triangleleft [x \mapsto t]\ S_2 \rangle\ (s\ t)$ is equivalent to $s\ t$, hence the cast is behaviorally equivalent to the identity function. □

Since cast insertion only inserts upcasts on well-typed programs, it follows that cast insertion preserves extensional and contextual equivalence for such programs.

Corollary 15.

(1) *If* $E \vdash t : T$ *and* $E \vdash t \hookrightarrow t' : T$*, then* $E \vdash t \sim t' : T$

(2) *If* $E \vdash T$ *and* $E \vdash T \hookrightarrow T'$*, then* $E \vdash T <: T'$ *and* $E \vdash T' <: T$

Proof. By mutual induction on the derivations of $E \vdash t : T$ and $E \vdash T$. In each case, because algorithmic subtyping can only return ? or $\surd$, the output of cast insertion is simply the input with redundant casts inserted. By Theorem 14 these are all equivalent to identity functions. □

It follows that cast insertion accepts all well-typed programs. Since casts are inserted into types during this process, part (2) of Corollary 15 is crucial to ensuring that the meaning of well-formed types does not change.

Theorem 16 (Compilation Completeness). *Suppose* $\vdash E$ *and that* s *and* T *contain no casts.*

(1) *If* $E \vdash s : S$ *then* $\exists t, T$ *such that* $E \vdash s \hookrightarrow t : T$*.*

(2) *If* $E \vdash S$ *then* $\exists T$ *such that* $E \vdash S \hookrightarrow T$*.*

Proof. By induction on the derivations. □

7. STATIC CHECKING VS. HYBRID CHECKING

Given the proven benefits of traditional, purely-static type systems, an important question that arises is how hybrid type checkers relate to static type checkers.

To study this question, suppose we are given a static type checker that targets a restricted subset of λ^H for which type checking is statically decidable. Specifically, suppose $\mathcal{D}$ is a subset of *Term* such that for all $t_1, t_2 \in \mathcal{D}$ and for all singleton environments $x : B$, the judgment $x : B \vdash t_1 \Rightarrow t_2$ is decidable. We introduce a statically-decidable language λ^S that is obtained from λ^H by only permitting $\mathcal{D}$ predicates in refinement types. We also assume all types in λ^S are closed, to avoid the complications of substituting arbitrary terms into refinement predicates via the rule [T-App] (and hence the above environment $x : B$ suffices). It then follows that subtyping and type checking for λ^S are decidable, and we denote this type checking judgment as $E \vdash^{\mathsf{s}} t : T$.

As an extreme, we could take $\mathcal{D} = \{\texttt{true}\}$, in which case the λ^S type language is essentially the simply typed λ-calculus:

$$T ::= B \mid T \to T$$

However, to yield a more general argument, we assume only that $\mathcal{D}$ is a subset of *Term* for which implication is decidable.

Clearly, the hybrid implication algorithm can give precise answers on (decidable) $\mathcal{D}$-terms, and so we assume that for all $t_1, t_2 \in \mathcal{D}$ and for all environments $x : B$, the judgment $x : B \vdash^a_{alg} t_1 \Rightarrow t_2$ holds for some $a \in \{\surd, \times\}$. Under this assumption, hybrid type checking behaves identically to static type checking on (well-typed or ill-typed) λ^S programs.

Theorem 17. *For all λ^S terms t, λ^S environments E, and λ^S types T, the following three statements are equivalent:*

(*1*) $E \vdash^{\mathsf{S}} t : T$
(*2*) $E \vdash t : T$
(*3*) $E \vdash t \hookrightarrow t : T$

Proof. The hybrid implication algorithm is complete on $\mathcal{D}$-terms, and hence the hybrid subtyping algorithm is complete for λ^S types. The proof then follows by induction on typing derivations. $\Box$

Thus, to a λ^S programmer, a hybrid type checker behaves exactly like a traditional static type checker.

We now compare static and hybrid type checking from the perspective of a λ^H programmer. To enable this comparison, we need to map expressive λ^H types into the more restrictive λ^S types, and in particular to map arbitrary boolean predicates into $\mathcal{D}$ predicates. We assume the computable function

$$\gamma : \mathit{Term} \rightarrow \mathcal{D}$$

performs this mapping. The function *erase* then maps λ^H refinement types to λ^S refinement types by using γ to abstract boolean terms:

$$\mathit{erase}(\{x{:}B \mid t\}) \stackrel{\text{def}}{=} \{x{:}B \mid \gamma(t)\}$$

We extend *erase* in a compatible manner to map λ^H types, terms, and environments to corresponding λ^S types, terms, and environments. Thus, for any λ^H program P, this function yields the corresponding λ^S program $\mathit{erase}(P)$.

As might be expected, the *erase* function must lose information, and we now explore the consequences of this information loss for programs with complex specifications. As an extreme example, let $\mathit{Halt}_{n,m}$ denote a closed formula that encodes "Turing machine m eventually halts on input n". Suppose that γ maps $\mathit{Halt}_{n,m}$ to `false` for all n and m, and consider the collection of programs $P_{n,m}$, which includes both well-typed and ill-typed programs:

$$P_{n,m} \stackrel{\text{def}}{=} (\lambda x{:}\{x{:}\texttt{Int} \mid \mathit{Halt}_{n,m}\}.\, x)\ 1$$

Decidability arguments show that the hybrid type checker will accept some ill-typed program $P_{n,m}$ based on its inability to statically prove that $\mathit{Halt}_{n,m} = \texttt{false}$. In contrast, the static type checker will reject (the erased version of) $P_{n,m}$. Hence:

The static type checker statically rejects (the erased version of) an ill-typed program that the hybrid type checker accepts.

Thus, the static type checker performs better in this situation.

Conversely, this particular static type checker will also reject (the erased version of) many well-typed programs $P_{n,m}$ that the hybrid type checker accepts. The following theorem generalizes this argument, and shows that for any computable mapping γ there exists some program P such that hybrid type checking of P performs better than static type checking of *erase*(P).

THEOREM 18. *For any computable mapping γ either:*

(1) *the static type checker rejects the erased version of some well-typed λ^H program, or*

(2) *the static type checker accepts the erased version of some ill-typed λ^H program for which the hybrid type checker would statically detect the error.*

PROOF. Let E be the environment $x : \texttt{Int}$.

By reduction from the halting problem, the judgment $E \vdash t \Rightarrow \texttt{false}$ for arbitrary boolean terms t is undecidable. However, the implication judgment $E \vdash \gamma(t) \Rightarrow \gamma(\texttt{false})$ is decidable. Hence these two judgments are not equivalent, *i.e.*:

$$\{t \mid (E \vdash t \Rightarrow \texttt{false})\} \quad \neq \quad \{t \mid (E \vdash \gamma(t) \Rightarrow \gamma(\texttt{false}))\}$$

It follows that there must exists some *witness* w that is in one of these sets but not the other, and so one of the following two cases must hold.

(1) Suppose:

$$\begin{array}{l} E \vdash w \Rightarrow \texttt{false} \\ E \not\vdash \gamma(w) \Rightarrow \gamma(\texttt{false}) \end{array}$$

We construct as a counter-example the program P_1:

$$P_1 \quad \stackrel{\text{def}}{=} \quad \lambda x{:}\{x{:}\texttt{Int} \mid w\}.\, (x \texttt{ as } \{x{:}\texttt{Int} \mid \texttt{false}\})$$

From the assumption $E \vdash w \Rightarrow \texttt{false}$ the subtyping judgment

$$\emptyset \vdash \{x{:}\texttt{Int} \mid w\} <: \{x{:}\texttt{Int} \mid \texttt{false}\}$$

holds. Hence, P_1 is well-typed, and (by Lemma 16) is accepted by the hybrid type checker. However, from the assumption $E \not\vdash \gamma(w) \Rightarrow \gamma(\texttt{false})$ the erased version of the subtyping judgment does not hold:

$$\emptyset \not\vdash \mathit{erase}(\{x{:}\texttt{Int} \mid w\}) <: \mathit{erase}(\{x{:}\texttt{Int} \mid \texttt{false}\})$$

Hence *erase*(P_1) is ill-typed and rejected by the static type checker.

(2) Conversely, suppose:

$$\begin{array}{l} E \not\vdash w \Rightarrow \texttt{false} \\ E \vdash \gamma(w) \Rightarrow \gamma(\texttt{false}) \end{array}$$

From the first supposition and by the definition of the implication judgment, there exists integers n and m such that

$$[x \mapsto n]\, w \leadsto^m \texttt{true}$$

We now construct as a counter-example the program P_2:

$$P_2 \quad \overset{\text{def}}{=} \quad \lambda x{:}\{x{:}\texttt{Int} \,|\, w\}.\,(x \texttt{ as } \{x{:}\texttt{Int} \,|\, \texttt{false} \wedge (n = m)\})$$

In the program P_2, the term $n = m$ has no semantic meaning since it is conjoined with `false`. The purpose of this term is to serve only as a "hint" to the following rule for refuting implications (which we assume is included in the reasoning performed by the implication algorithm). In this rule, the integers a and b serve as hints, and take the place of randomly generated values for testing if t ever evaluates to `true`.

$$\frac{[x \mapsto a]\, t \leadsto^{b} \texttt{true}}{E \vdash^{\times}_{alg} t \Rightarrow (\texttt{false} \wedge a = b)}$$

This rule enables the implication algorithm to conclude that:

$$E \vdash^{\times}_{alg} w \Rightarrow \texttt{false} \wedge (n = m)$$

Hence, the subtyping algorithm can conclude:

$$\vdash^{\times}_{alg} \{x{:}\texttt{Int} \,|\, w\} <: \{x{:}\texttt{Int} \,|\, \texttt{false} \wedge (n = m)\}$$

Therefore, the hybrid type checker rejects P_2, which by Lemma 16 is therefore ill-typed.

$$\forall P, T. \quad \not\vdash P_2 \hookrightarrow P : T$$

We next consider how the static type checker behaves on the program *erase*(P_2). We consider two cases, depending on whether the following implication judgement holds:

$$E \vdash \gamma(\texttt{false}) \Rightarrow \gamma(\texttt{false} \wedge (n = m))$$

(a) If this judgment holds then by the transitivity of implication and the assumption $E \vdash \gamma(w) \Rightarrow \gamma(\texttt{false})$ we have that:

$$E \vdash \gamma(w) \Rightarrow \gamma(\texttt{false} \wedge (n = m))$$

Hence the subtyping judgement

$$\emptyset \vdash \{x{:}\texttt{Int} \,|\, \gamma(w)\} <: \{x{:}\texttt{Int} \,|\, \gamma(\texttt{false} \wedge (n = m))\}$$

holds and the program *erase*(P_2) is accepted by the static type checker:

$$\emptyset \vdash \mathit{erase}(P_2) : \{x{:}\texttt{Int} \,|\, \gamma(w)\} \rightarrow \{x{:}\texttt{Int} \,|\, \gamma(\texttt{false} \wedge (n = m))\}$$

(b) If the above judgment does not hold then consider as a counter-example the program P_3:

$$P_3 \overset{\text{def}}{=} \lambda x{:}\{x{:}\texttt{Int} \,|\texttt{false}\}.\,(x \texttt{ as } \{x{:}\texttt{Int} \,|\, \texttt{false} \wedge (n{=}m)\})$$

This program is well-typed, from the subtype judgment:

$$\emptyset \vdash \{x{:}\texttt{Int} \,|\, \texttt{false}\} <: \{x{:}\texttt{Int} \,|\, \texttt{false} \wedge (n = m)\}$$

However, the erased version of this subtype judgment does not hold:

$$\emptyset \not\vdash \mathit{erase}(\{x{:}\texttt{Int} \,|\texttt{false}\}) <: \mathit{erase}(\{x{:}\texttt{Int} \,|\, \texttt{false} \wedge (n{=}m)\})$$

Hence, $erase(P_3)$ is rejected by the static type checker:

$$\forall T. \quad \emptyset \not\vdash^{s} erase(P_3) \,:\, T$$

□

8. RELATED WORK

Much prior work has focused on dynamic checking of expressive specifications, or *contracts* [D. L. Parnas 1972; Meyer 1988; Holt and Cordy 1988; Luckham 1990; Gomes et al. 1996; Kölling and Rosenberg 1997; Findler and Felleisen 2002; Leavens and Cheon 2005; Blume and McAllester 2006]. An entire design philosophy, *Contract Oriented Design*, has been based on dynamically-checked specifications. Hybrid type checking embraces precise specifications, but extends prior purely-dynamic techniques to verify (or detect violations of) expressive specifications statically, wherever possible.

The programming language Eiffel [Meyer 1988] 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 somewhat awkward, since it requires the programmer to factor each specification into its static and dynamic components. Furthermore, the factoring is too rigid, since the specification needs to be manually refactored to exploit improvements in static checking technology.

Other authors have considered pragmatic combinations of both static and dynamic checking. Abadi, Cardelli, Pierce and Plotkin [Abadi, M., L. Cardelli, B. Pierce, and G. Plotkin 1989] extended a static type system with a type `Dynamic` that could be explicitly cast to and from any other type (with appropriate run-time checks). Henglein characterized the *completion process* of inserting the necessary coercions, and presented a rewriting system for generating minimal completions [Henglein 1994]. Thatte developed a similar system in which the necessary casts are implicit [Thatte, S. 1990]. These systems are 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 large range of specifications, from `Dynamic` at one end to precise hybrid-checked specifications at the other.

Research on advanced type systems has influenced our choice of how to express program invariants. In particular, Freeman and Pfenning [Freeman and Pfenning 1991] extended ML with another form of refinement types. They do not support arbitrary refinement predicates, since their system provides both decidable type checking and type inference. Xi and Pfenning have explored the practical application of dependent types in an extension of ML called Dependent ML [Xi and Pfenning 1999; Xi 2000]. Decidability of type checking is preserved by appropriately restricting which terms can appear in types. Despite these restrictions, a number of interesting examples can be expressed in Dependent ML.

Ou, Tan, Mandelbaum, and Walker developed a type system similar to ours that combines dynamic checks with refinement and dependent types [Ou et al. 2004]. Unlike hybrid type checking approach, their type system restricts refinement predicates to ensure decidability, and it supports mutable data. In addition, whereas hybrid type checking uses dynamic checks to circumvent decidability limitations, their system uses dynamic checks to permit interoperability between precisely typed code fragments (that use refinement types) and more coarsely typed code fragments (that do not), and so it permits the introduction of refinement predicates into a large program in an incremental manner.

There has been much recent work on alternative forms of hybrid and gradual type systems. Siek and Taha independently developed a system of *gradual typing* [Siek and Taha 2006; 2007] that combines dynamic and static typing via casts similar to ours. However, they do not address refinement predicates, and so they avoid much of the complexities of hybrid type checking.

Gronski *et al* [Gronski et al. 2006] developed a prototype language that used hybrid type checking to blend refinement types with dynamic typing, records, variants, and first-class types. Their experiments show that for many common examples, the number of inserted casts is small or none.

Wadler and Findler prove that the more precisely typed portion of a program cannot be blamed for a runtime failure [Wadler and Findler 2007]. This technique may furnish yet another proof of behavioral correctness for hybrid type checking, as it ensures that any redundant cast cannot fail at run time.

Refinement types, in addition to causing a burden in terms of challenging proof obligations, impose an annotation burden on the programmer. Knowles and Flanagan [Knowles and Flanagan 2007] present a theoretical closed-form solution to the type inference problem for general refinement types. Rondon *et al* [Rondon et al. 2008] have subsequently developed an implementation for OCaml using abstract interpretation to find approximations to these closed-form solutions.

The static checking tool ESC/Java [Flanagan et al. 2002] checks expressive JML specifications [Burdy et al. 2003; Leavens and Cheon 2005] using the Simplify automatic theorem prover [Detlefs et al. 2005]. However, Simplify does not distinguish between failing to prove a theorem and finding a counter-example that refutes the theorem, and so ESC/Java's error messages may be caused either by incorrect programs or by limitations in its theorem prover.

The limitations of purely-static and purely-dynamic approaches have also motivated other work on hybrid analyses. For example, CCured [Necula et al. 2002] is a sophisticated hybrid analysis for preventing the ubiqutous array bounds violations in the C programming language. 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 data race condition checking [von Praun and Gross 2001; O'Callahan and Choi 2003; Agarwal and Stoller 2004].

Prior work (*e.g.* [Breazu-Tannen et al. 1991]) introduced and studied implicit coercions in type systems. Note that there are no implicit coercions in the λ^H type system itself, but only in the cast insertion algorithm, and so we do not need a coherence theorem for λ^H, but instead reason about the connection between the type system and cast insertion algorithm.

9. CONCLUSIONS AND FUTURE WORK

Precise specifications are essential for modular software development. Hybrid type checking suggests an interesting approach for providing high coverage checking of precise specifications. This paper explores hybrid type checking in the idealized context of the λ-calculus, and highlights some of the key principles and implications of hybrid type checking.

In terms of software deployment, an important topic is recovery methods for post-deployment cast failures; transactional roll-back mechanisms [Haines et al. 1994; Vitek et al. 2004] may be useful in this regard. Hybrid type checking may also allow precise types to be preserved during the compilation and distribution process, via techniques such as proof-carrying code [Necula 1997] and typed assembly language [Morrisett et al. 1999].

Since the introduction of hybrid type checking, several important practical aspects have been explored, such as type inference [Knowles and Flanagan 2007; Rondon et al. 2008], space efficiency [Herman et al. 2007], and interaction with realistic programming features such as records, variants, and objects [Gronski et al. 2006]. The most pressing topic that remains to be investigated is the effectiveness of contract types, and hybrid type checking in particular, in large-scale realistic software development.

Acknowledgements Thanks to Matthias Felleisen, Stephen Freund, Robby Findler, Martín Abadi, Shriram Krishnamurthi, David Walker, Aaron Tomb, and Jessica Gronski for valuable feedback on this paper. This work was supported by the National Science Foundation under Grants CCR-0341179 and CCR-0707885, and by a Fellowship from the Sloan Foundation.

REFERENCES

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

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

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

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

Augustsson, L. 1998. Cayenne — a language with dependent types. In *Proceedings of the ACM International Conference on Functional Programming*. 239–250.

Blei, D., Harrelson, C., Jhala, R., Majumdar, R., Necula, G. C., Rahul, S. P., Weimer, W., and Weitz, D. 2000. Vampyre. Information available from `http://www-cad.eecs.berkeley.edu/~rupak/Vampyre/`.

Blume, M. and McAllester, D. 2006. Sound and complete models for contracts. *Journal of Functional Programming 16*, 375 – 414.

Breazu-Tannen, V., Coquand, T., Gunter, C. A., and Scedrov, A. 1991. Inheritance as implicit coercion. *Inf. Comput. 93,* 1, 172–221.

Burdy, L., Cheon, Y., Cok, D., Ernst, M., Kiniry, J., Leavens, G., Leino, K., and Poll, E. 2003. An overview of JML tools and applications.

Cardelli, L. 1988a. Phase distinctions in type theory. Manuscript.

Cardelli, L. 1988b. Typechecking dependent types and subtypes. In *Lecture notes in computer science on Foundations of logic and functional programming*. 45–57.

Davies, R. and Pfenning, F. 2000. Intersection types and computational effects. In *ICFP '00: fifth ACM SIGPLAN international conference on Functional programming*. 198–208.

Denney, E. 1998. Refinement types for specification. In *Proceedings of the IFIP International Conference on Programming Concepts and Methods*. Vol. 125. Chapman & Hall, 148–166.

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

Fagan, M. 1990. Soft typing. Ph.D. thesis, Rice University.

Findler, R. B. 2002. Behavioral software contracts. Ph.D. thesis, Rice University.

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

Flanagan, C. 2006. Hybrid type checking. In *Symposium on Principles of Programming Languages*. 245 – 256.

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

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

Freeman, T. and Pfenning, F. 1991. Refinement types for ML. In *Conference on Programming Language Design and Implementation*. 268–277.

Frisch, A., Castagna, G., and Benzaken, V. 2002. Semantic subtyping. 137–146.

Godefroid, P., Klarlund, N., and Sen, K. 2005. DART: Directed automated random testing. In *Conference on Programming Language Design and Implementation*.

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

Gosling, J., Joy, B., and Steele, G. 1996. *The Java Language Specification*. Addison-Wesley.

Gronski, J. and Flanagan, C. 2007. Unifying hybrid types and contracts. In *Trends in Functional Programming*.

Gronski, J., Knowles, K., Tomb, A., Freund, S. N., and Flanagan, C. 2006. Sage: Practical hybrid checking for expressive types and specifications. In *Proceedings of the Workshop on Scheme and Functional Programming*. 93–104.

Haines, N., Kindred, D., Morrisett, J. G., Nettles, S., and Wing, J. M. 1994. Composing first-class transactions. In *ACM Transactions on Programming Languages and Systems*. Vol. 16(6). 1719–1736.

Henglein, F. 1994. Dynamic typing: Syntax and proof theory. *Science of Computer Programming 22,* 3, 197–230.

Herman, D., Tomb, A., and Flanagan, C. 2007. Space-efficient gradual typing. In *Trends in Functional Programming*. 404–419.

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

Knowles, K. and Flanagan, C. 2007. Type reconstruction for general refinement types. In *European Symposium on Programming*.

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

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

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

Mandelbaum, Y., Walker, D., and Harper, R. 2003. An effective theory of type refinements. In *International Conference on Functional Programming*. ACM Press, New York, NY, USA, 213–225.

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

Morrisett, G., Walker, D., Crary, K., and Glew, N. 1999. From System F to typed assembly language. *ACM Transactions on Programming Languages and Systems 21,* 3, 527–568.

NECULA, G. C. 1997. Proof-carrying code. In *Symposium on Principles of Programming Languages.*

NECULA, G. C., MCPEAK, S., AND WEIMER, W. 2002. CCured: type-safe retrofitting of legacy code. In *Symposium on Principles of Programming Languages.* 128–139.

O'CALLAHAN, R. AND CHOI, J.-D. 2003. Hybrid dynamic data race detection. In *Symposium on Principles and Practice of Parallel Programming.* 167–178.

OU, X., TAN, G., MANDELBAUM, Y., AND WALKER, D. 2004. Dynamic typing with dependent types. In *Proceedings of the IFIP International Conference on Theoretical Computer Science.* 437–450.

RONDON, P. M., KAWAGUCHI, M., AND JHALA, R. 2008. Liquid types. In *Conference on Programming Language Design and Implementation.*

SIEK, J. AND TAHA, W. 2007. Gradual typing for objects. 2–27.

SIEK, J. G. AND TAHA, W. 2006. Gradual typing for functional languages. In *Proceedings of the Workshop on Scheme and Functional Programming.*

STATMAN, R. 1985. Logical relations and the typed lambda-calculus. *Information and Control 65,* 2/3, 85–97.

TARDITI, D., MORRISETT, G., CHENG, P., STONE, C., HARPER, R., AND LEE, P. 1996. TIL: A type-directed optimizing compiler for ML. *ACM SIGPLAN Notices 31,* 5, 181–192.

THATTE, S. 1990. Quasi-static typing. In *Symposium on Principles of Programming Languages.*

VITEK, J., JAGANNATHAN, S., WELC, A., AND HOSKING, A. L. 2004. A semantic framework for designer transactions. In *European Symposium on Programming.* 249–263.

VON PRAUN, C. AND GROSS, T. 2001. Object race detection. In *Conference on Object-Oriented Programming, Systems, Languages and Applications.* 70–82.

WADLER, P. AND FINDLER, R. B. 2007. Well-typed programs can't be blamed. In *Proceedings of the Workshop on Scheme and Functional Programming.*

WRIGHT, A. AND CARTWRIGHT, R. 1994. A practical soft type system for scheme. In *Conference on Lisp and Functional Programming.* 250–262.

WRIGHT, A. AND FELLEISEN, M. 1994. A syntactic approach to type soundness. *Info. Comput. 115,* 1, 38–94.

XI, H. 2000. Imperative programming with dependent types. In *LICS 2000.* 375–387.

XI, H. AND PFENNING, F. 1999. Dependent types in practical programming. In *POPL '99: 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages.* ACM Press, 214–227.