# Hybrid Type Checking

Cormac Flanagan

Department of Computer Science
University of California, Santa Cruz
cormac@cs.ucsc.edu

## Abstract

Traditional static type systems are very effective for verifying basic interface specifications, but are somewhat limited in the kinds specifications they support. Dynamically-checked contracts can enforce 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 simply-typed $\lambda$-calculus 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

***Keywords*** Type systems, contracts, static checking, dynamic checking

## 1. Motivation

The construction of reliable software is extremely difficult. For large systems, it requires a modular development strategy that, ideally, is based on precise and trusted interface specifications. In practice, however, programmers typically work in the context of a large collection of APIs whose behavior is only informally and imprecisely specified and understood. Practical mechanisms for specifying and verifying precise, behavioral aspects of interfaces are clearly needed.

Static type systems have proven to be extremely effective and practical tools for specifying and verifying basic interface specifications, and are widely adopted by most software engineering projects. However, traditional type systems are somewhat limited in the kinds of specifications they support. Ongoing research on more powerful type systems (*e.g.*, [45, 44, 17, 29, 11]) attempts to overcome some of these restrictions, via advanced features such as dependent and refinement types. 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* [30, 14, 26, 19, 24, 27, 36, 25] provides a simple method for checking more expressive specifications. Dynamic checking can easily support precise specifications, such as:

- Subrange types, *e.g.*, the function `printDigit` requires an integer in the range [0,9].
- Aliasing restrictions, *e.g.*, `swap` requires that its arguments are distinct reference cells.
- Ordering restrictions, *e.g.*, `binarySearch` requires that its argument is a sorted array.
- Size specifications, *e.g.*, 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 [39]) might naturally require that its input be well-typed, *i.e.*, that it satisfies the predicate `wellTyped : Expr → 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 to be incompatible in practice.[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 checking 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* provides a potential solution to the limitations of purely-static and purely-dynamic approaches.

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

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

[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 this approach is too heavyweight for widespread use.



*POPL'06* January 11–13, 2006, Charleston, South Carolina, USA.

| Ill-typed programs | | Well-typed programs | |
|---|---|---|---|
| Clearly ill-typed | Subtle programs | | Clearly well-typed |
| Rejected | Accepted with casts | | Accepted without casts |
| | Casts may fail | Casts never fail | |

**Figure 1.** Hybrid type checking on various programs.

This rule uses the antecedent $E \vdash S <: T$ to check 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 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.

Hence, we argue that the most satisfactory 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* [9]. Instead, our hybrid type checking approach transforms the above application into the code

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

where the additional *type cast* or *coercion* $\langle S \rhd T \rangle\ t_2$ dynamically checks that the value produced by $t_2$ is in the domain type $T$. Note that hybrid type checking supports very 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.

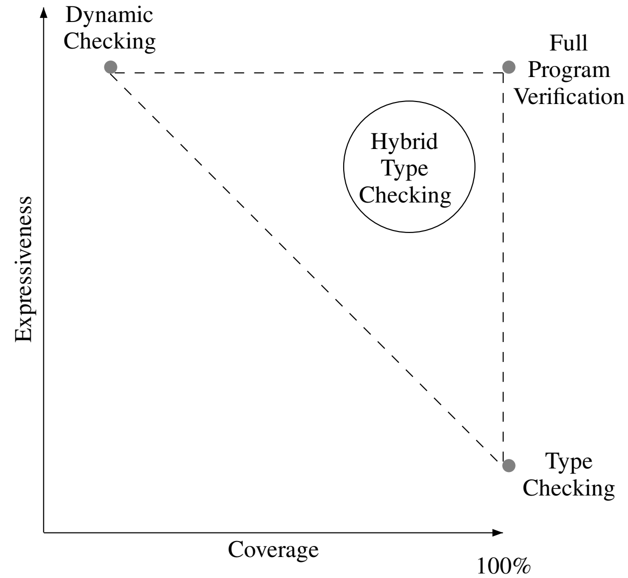


**Figure 2.** Rough sketch of the relationship between hybrid type checking, dynamic checking, type checking, and full program verification.

Hybrid type checking provides several desirable characteristics:

1. It supports precise interface specifications, which are essential for modular development of reliable software.
2. As many defects as is 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 never occur; they 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 compiled programs are behaviorally equivalent (or *bisimilar*).

Figure 2 contains a rough sketch of the relationship between hybrid type checking and prior approaches for program checking, in terms of expressiveness (y-axis) and coverage (x-axis). Dynamic checking is expressive but obtains limited coverage. Type checking obtains full coverage but has somewhat limited expressiveness. In theory, full program verification could provide full coverage for expressive specifications, but it is intractable for all but small programs. Motivated by the need for more expressive specification languages, the continuum between type checking and full program verification is being explored by a range of research projects (see, for example, [37, 5, 23, 45]). The goal of this paper is to investigate the interior of the triangle defined by these three extremes.

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. Moreover, as we shall see, for any decidable static type checker $S$, it is possible to develop a hybrid type checker $H$ that performs somewhat better than $S$ in the following sense:

1. $H$ dynamically detects errors that would be missed by $S$, since $H$ supports more precise specifications than $S$ and can detect violations of these specifications dynamically.
2. $H$ statically detects all errors that would be detected by $S$, since $H$ can statically perform the same reasoning as $S$.
3. $H$ actually detects more errors statically than $S$, since $H$ supports more precise specifications, and could reasonably detect some violations of these precise specifications statically.

The last property is perhaps the most surprising; Section 6 contains a proof that clarifies this argument.

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 compiler 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 a compiler-inserted cast $\langle S \rhd T\rangle\ v$, the value $v$ is a witness that refutes an undecided subtyping query, 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 compiler's analysis.

Indeed, just like different compilers for the same language may yield object code of different 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 [28, 42, 3, 15], 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` [1, 38], and in systems that combine dynamic checks with dependant types [35]. Hybrid type checking extends these ideas to support more precise specifications.

The general approach of hybrid type checking appears to be applicable to a variety of programming languages and to various specification languages. In this paper, we illustrate the key ideas of hybrid type checking for a fairly expressive dependent type system that is statically undecidable. Specifically, we work with an extension of the simply-typed $\lambda$-calculus that supports arbitrary refinements of base types.

This language and type system is described in the following section. Section 3 then presents a hybrid type checking algorithm for this language. Section 4 illustrates this algorithm on an example program. Section 5 verifies key correctness properties of our language and compilation algorithm. Section 6 performs a detailed comparison of the static and hybrid approaches to type checking. Section 7 discusses related work, and Section 8 describes opportunities for future research.

**Figure 3: Syntax**

| | | |
|---|---|---|
| $s, t ::=$ | | *Terms:* |
| | $x$ | variable |
| | $c$ | constant |
| | $\lambda x\!:\!S.\, t$ | abstraction |
| | $(t\ t)^l$ | application |
| | $\langle S \rhd T\rangle^l\ t$ | type cast |
| $S, T ::=$ | | *Types:* |
| | $x\!:\!S \to T$ | 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 |

## 2. The Language $\lambda^H$

This section introduces a variant of the simply-typed $\lambda$-calculus extended with casts and with precise (and hence undecidable) refinement types. We refer to this language as $\lambda^H$.

### 2.1 Syntax of $\lambda^H$

The syntax of $\lambda^H$ is summarized in Figure 3. Terms include variables, constants, functions, applications, and casts . The cast $\langle S \rhd T\rangle\ t$ dynamically checks that the result of $t$ is of type $T$ (in a manner similar to coercions [38], contracts [13, 14], and to type casts in languages such as Java [20]). For technical reasons, the cast also includes that static type $S$ of the term $t$. Type casts are annotated with associated labels $l \in$ *Label*, which are used to map run-time errors back to locations in the source program. Applications are also annotated with labels, for similar reasons. For clarity, we omit these labels when they are irrelevant.

The $\lambda^H$ type language includes dependent function types [10], for which we use the syntax $x : S \to T$ of Cayenne [4] (in preference to the equivalent syntax $\Pi x : S.\ T$). 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 *base refinement types* of 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 $t[x := c]$ evaluates to true. 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 $\{x\!:\!B \mid \mathtt{true}\}$.

Our refinement types are inspired by prior work on decidable refinement type systems [29, 17, 11, 45, 44, 35]. However, our refinement types support arbitrary predicates, and this expressive power causes type checking to become undecidable. For example, 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* → *Type*, and the set *Constant* is implicitly defined as the domain of this mapping.

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

A *basic constant* is a constant whose type is a base refinement 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:

$$\begin{array}{l}
\texttt{fix}_{\texttt{Int}\to\texttt{Int}} \\
\quad \lambda f{:}(\texttt{Int} \to \texttt{Int}).\ \lambda n{:}\texttt{Int}. \\
\qquad \texttt{if}_{\texttt{Int}}\ (n = 0)\ \ 1\ \ (n * (f\ (n-1)))
\end{array}$$

Refinement types can express many precise specifications, such as:

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

Here, we assume that `Unit`, `Array`, and `RefInt` are additional base types, and the primitive function $\texttt{sorted} : \texttt{Array} \to \texttt{Bool}$ identifies sorted arrays.

### 2.2 Operational Semantics of $\lambda^H$

We next describe the run-time behavior of $\lambda^H$ terms, since the semantics of the type language depends on the semantics of terms. The relation $s \longrightarrow t$ performs a single evaluation step, and the relation $\longrightarrow^*$ is the reflexive-transitive closure of $\longrightarrow$.

As shown in Figure 4, the rule [E-$\beta$] performs standard $\beta$-reduction of function applications. The rule [E-PRIM] evaluates applications of primitive functions. This rule is defined in terms of the partial function:

$$[\![\cdot]\!]\ \cdot : \textit{Constant} \times \textit{Term} \to_p \textit{Term}$$

**Figure 4: Evaluation Rules**

Evaluation $\boxed{s \longrightarrow t}$

$$(\lambda x{:}S.\ t)\ s \quad\longrightarrow\quad t[x := s] \qquad [\text{E-}\beta]$$

$$c\ t \quad\longrightarrow\quad [\![c]\!](t) \qquad [\text{E-PRIM}]$$

$$\langle (x{:}S_1 \to S_2) \rhd (x{:}T_1 \to T_2)\rangle^l\ t \quad\longrightarrow\quad \lambda x{:}T_1.\ \langle S_2 \rhd T_2\rangle^l\ (t\ (\langle T_1 \rhd S_1\rangle^l\ x)) \qquad [\text{E-CAST-F}]$$

$$\langle \{x{:}B \mid s\} \rhd \{x{:}B \mid t\}\rangle\ c \quad\longrightarrow\quad c \quad \text{if } t[x := c] \longrightarrow^* \texttt{true} \qquad [\text{E-CAST-C}]$$

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

Contexts $\boxed{C}$

$$C \ ::= \ \bullet \mid \bullet\ t \mid t\ \bullet \mid \langle S \rhd T\rangle\ \bullet$$

which defines the semantics of primitive functions. For example:

$$\begin{array}{rcl}
[\![\texttt{not}]\!](\texttt{true}) & = & \texttt{false} \\
[\![+]\!](3) & = & +_3 \\
[\![+_3]\!](4) & = & 7 \\
[\![\texttt{not}]\!](3) & = & \textit{undefined} \\
[\![\texttt{if}_T]\!](\texttt{true}) & = & \lambda x{:}T.\ \lambda y{:}T.\ x \\
[\![\texttt{if}_T]\!](\texttt{false}) & = & \lambda x{:}T.\ \lambda y{:}T.\ y \\
[\![\texttt{fix}_T]\!](t) & = & t\ (\texttt{fix}_T\ t)
\end{array}$$

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

$$\lambda x{:}T_1.\ \langle S_2 \rhd T_2\rangle^l\ (t\ (\langle T_1 \rhd S_1\rangle^l\ x))$$

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 $t$, and the result of that application is then cast to the desired result type $T_2$. Thus, higher-order casts are performed a lazy fashion – the new casts $\langle S_2 \rhd T_2\rangle^l$ and $\langle T_1 \rhd S_1\rangle^l$ are performed at every application of the resulting function, in a manner reminiscent of higher-order contracts [14]. If either of the two new casts fail, their label $l$ then assigns the blame back to the original cast $\langle (x{:}S_1 \to S_2) \rhd (x{:}T_1 \to T_2)\rangle^l$.

The rule [E-CAST2] deals with casting a basic constant $c$ to a base refinement type $\{x{:}B \mid t\}$. This rule checks that the predicate $t$ holds on $c$, *i.e.*, that $t[x := c]$ evaluates to `true`.

Note that these casts involve only tag checks, predicate checks, and creating checking wrappers for functions. Thus, our approach adheres to the principle of phase separation [9], in that there is no type checking of actual program syntax at run time.

### 2.3 The $\lambda^H$ Type System

We next describe the (undecidable) $\lambda^H$ type system via the collection of type judgments and rules shown in Figure 5. 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 $E$.

**Figure 5: Type Rules**

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

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

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

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

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

$$\frac{E \vdash t : S \qquad E \vdash T}{E \vdash \langle S \rhd T \rangle\, t : T} \quad \text{[T-CAST]}$$

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

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

$$\frac{\forall \sigma.\, (E \models \sigma \text{ and } \sigma(s) \longrightarrow^* \texttt{true} \text{ implies } \sigma(t) \longrightarrow^* \texttt{true})}{E \vdash s \Rightarrow t} \quad \text{[IMP]}$$

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

$$\frac{}{\emptyset \models \emptyset} \quad \text{[CS-EMPTY]}$$

$$\frac{\emptyset \vdash t : T \qquad (x := t)E \models \sigma}{x : T, E \models (x := t, \sigma)} \quad \text{[CS-EXT]}$$

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]. We assume that variables are bound at most once in an environment. As customary, we apply implicit $\alpha$-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 refinement types. Recall that a type $\{x : B \mid t\}$ denotes the set of constants $c$ of type $B$ for which $t[x := c]$ evaluates to `true`. We use two auxiliary judgments to express the correct subtyping relation between refinement types. The implication judgment $E \vdash t_1 \Rightarrow t_2$ holds if whenever the term $t_1$ evaluates to `true` then $t_2$ also evaluates to `true`. This relation is defined in terms of substitutions that are *consistent* with $E$. Specifically, a substitution $\sigma$ (from variables to terms) is *consistent* with the environment $E$ if $\sigma(x)$ is of type $E(x)$ for each $x \in dom(E)$. Finally, the rule [S-BASE] states that the subtyping judgment $E \vdash \{x{:}B \mid t_1\} <: \{x{:}B \mid t_2\}$ holds if

$$E, x : B \vdash t_1 \Rightarrow t_2$$

meaning that every constant of type $\{x : B \mid t_1\}$ also has type $\{x{:}B \mid t_2\}$.

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:

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

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 $\lambda^H$

We now describe how to perform hybrid type checking for the language $\lambda^H$. We believe this general approach extends to other languages with similarly expressive type systems.

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 due to a timeout 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 6. 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 domain and covariant range types, via the rule [SA-ARROW]. This rule uses the following conjunction operation $\otimes$ between three-valued results:

| $\otimes$ | $\surd$ | ? | $\times$ |
|---|---|---|---|
| $\surd$ | $\surd$ | ? | $\times$ |
| ? | ? | ? | $\times$ |
| $\times$ | $\times$ | $\times$ | $\times$ |

**Figure 6: Compilation Rules**

Compilation of terms $\boxed{E \vdash s \hookrightarrow t : T}$

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

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

$$\frac{E \vdash S \hookrightarrow T \qquad E, x : T \vdash s \hookrightarrow t : T'}{E \vdash (\lambda x{:}S.\, s) \hookrightarrow (\lambda x{:}T.\, t) : (x{:}T \rightarrow T')} \quad \text{[C-FUN]}$$

$$\frac{E \vdash s_1 \hookrightarrow t_1 : (x{:}T \rightarrow T') \qquad E \vdash s_2 \hookrightarrow t_2 \downarrow^l T}{E \vdash (s_1\ s_2)^l \hookrightarrow (t_1\ t_2)^l : T'[x := t_2]} \quad \text{[C-APP]}$$

$$\frac{E \vdash S_1 \hookrightarrow T_1 \qquad E \vdash S_2 \hookrightarrow T_2 \qquad E \vdash s \hookrightarrow t \downarrow^l T_1}{E \vdash \langle S_1 \rhd S_2 \rangle^l\, s \hookrightarrow \langle T_1 \rhd T_2 \rangle^l\, t : T_2} \quad \text{[C-CAST]}$$

Compilation and checking $\boxed{E \vdash s \hookrightarrow t \downarrow^l T}$

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

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

Compilation of types $\boxed{E \vdash S \hookrightarrow T}$

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

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

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

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

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

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

separate algorithm

If the appropriate subtyping relation holds for certain 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 corresponding 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 may 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 *compilation 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 compilation process to types via the judgment $E \vdash S \hookrightarrow T$. Some of the compilation rules rely on the auxiliary *compilation and checking* judgment

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

This judgment takes as input an environment $E$, a source term $s$, and a desired result type $T$, and checks that $s$ compiles to a term of this result type. The label $l$ is used to appropriately annotate casts inserted by this compilation and checking process.

The rules defining these judgments are shown in Figure 6. 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] compiles an abstraction $\lambda x : S.\, t$ by compiling the type $S$ to $S'$ and compiling $t$ to $t'$ of type $T$, and then yielding the compiled abstraction $\lambda x : S'.\, t'$ of type $x : S' \rightarrow T$. The rule [C-APP] for an application $s_1\ s_2$ compiles $s_1$ to a term $t_1$ of function type $x : T \rightarrow T'$, and uses the compilation and checking judgment to ensure that the argument term $s_2$ compiles into a term of the appropriate argument type $T$. The rule [C-CAST] for a cast $\langle S_1 \rhd S_2 \rangle\ s$ compiles the two types $S_1$ and $S_2$ into $T_1$ and $T_2$, respectively, and then uses the compilation and checking judgment to ensure that $s$ compiles to a term $t$ of type expected type $T_1$.

The two rules defining the compilation and checking judgment $E \vdash s \hookrightarrow u \downarrow^l T$ demonstrate the key idea of hybrid type checking. Both rules start by compiling $s$ to a term $t$ of some type $S$. The crucial question is then whether this type $S$ is a subtype of the expect 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 $t$ as the compiled form of $s$.
- 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 compilation rule is applicable.
- Otherwise, in the uncertain case where $E \vdash^{?}_{alg} S <: T$, the rule [CC-CHK] inserts the type cast $\langle S \rhd T \rangle^l$ to dynamically ensure that values returned by $t$ are actually of the desired type $T$.

These rules for compilation 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.

Compilation of types is straightforward. The rule [C-ARROW] compiles a dependent function type $x : S \to T$ by recursively compiling the types $S$ and $T$ (in appropriate environments) to $S'$ and $T'$ respectively, and then yielding the compiled function type $x : S' \to T'$. The rule [C-BASE] compiles a base refinement type $\{x : B \mid t\}$ by compiling the term $t$ to $t'$ (whose type should be a subtype of `Bool`), and then yielding the compiled base refinement type $\{x : B \mid t'\}$.

Note that checking that a type is well-formed is actually a compilation process that returns a well-formed type (possibly with added casts). Thus, we only perform compilation of types where necessary, at $\lambda$-abstractions and casts, when we encounter (possibly ill-formed) types in the source program. In particular, the compilation rules do not explicitly check that the environment is well-formed, since that would involve repeatedly compiling all types in that environment. Instead, the compilation 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 hybrid compilation algorithm, consider a function `serializeMatrix` that serializes an $n$ by $m$ matrix into an array of size $n \times m$. We extend the language $\lambda^H$ 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:

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

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

$$\begin{array}{rcl} \texttt{Array}_n & \stackrel{\text{def}}{=} & \{a{:}\texttt{Array} \mid (\texttt{asize}\, a = n)\} \\ \texttt{Matrix}_{n,m} & \stackrel{\text{def}}{=} & \{a{:}\texttt{Matrix} \mid \left(\begin{array}{r} \texttt{matrixWidth}\;\; a = n \\ \wedge\, \texttt{matrixHeight}\, a = m \end{array}\right)\} \end{array}$$

The shorthand $t\ \texttt{as}^l\ T$ ensures that the term $t$ has type $T$ by passing $t$ as an argument to the identity function of type $T \to T$:

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

We now define the function `serializeMatrix` as:

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

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`:

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

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:

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

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.*, [12, 6]) 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 the program compiles without any additional casts.
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 multiplications 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 compiled function (where due to space constraints we have elided the source type of the cast):

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

   This term can be optimized, via [E-$\beta$] and [E-CAST-F] steps and via removal of clearly redundant $\langle \texttt{Int} \rhd \texttt{Int}\rangle$ casts, to:

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

   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:

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

   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 [18] 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 compiled program:

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

   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. Moreover,

**Figure 7: 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 cast label $l$ will identify the $\texttt{as}^l$ construct in the original program as the location of this type error, thus indicating that the original definition of `serializeMatrix` did not satisfy its specification.

Note that prior work on practical dependent types [45] could not handle any of 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 clearly an improvement over not detecting these errors at all, and getting subsequent crashes or incorrect results.

## 5. Correctness

We now study what correctness properties are guaranteed by hybrid type checking, starting with the type system, which provides the specification for our hybrid compilation algorithm.

### 5.1 Correctness of the Type System

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 $\lambda$-abstraction 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:*

- *$c$ has a well-formed type,* i.e. $\emptyset \vdash ty(c)$.
- *If $c$ is a primitive function then it cannot get stuck and its operational behavior is compatible with its type,* i.e.
  - *if $E \vdash c\ v\ :\ T$ then $[\![c]\!](v)$ is defined*
  - *if $E \vdash c\ t\ :\ T$ and $[\![c]\!](t)$ is defined then $E \vdash [\![c]\!](t)\ :\ T$.*
- *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 $t[x := c] \longrightarrow^* \texttt{true}$*
  - *if $ty(c) = \{x\!:\!B \mid t\}$ then $\forall c' \in$ Constant. $t[x := c'] \longrightarrow^* \texttt{true}$ implies $c = c'$*

The type system satisfies the following type preservation or subject reduction property [43]. This theorem includes a requirement that the environment $E$ is well-formed ($\vdash E$), a notion that is defined in Figure 7. Note that the type rules do not refer to this judgment directly in order to yield a closer correspondence with the compilation rules.

THEOREM 2 (Preservation).
*If $\vdash E$ and $E \vdash s\ :\ T$ and $s \longrightarrow t$ then $E \vdash t\ :\ T$.*

PROOF: By induction on the typing derivation $E \vdash s\ :\ T$, based on the usual substitution lemma. $\square$

The type system also satisfies the progress property, with the caveat that type casts may fail. A failed cast is one that either (1) casts a basic constant to a function type, (2) casts a function to a base refinement type, or (3) casts a constant to an incompatible refinement type (*i.e.*, one with a different base type or an incompatible predicate)

DEFINITION 3 (Failed Casts). *A* failed cast *is one of:*

1. $\langle \{x\!:\!B \mid s\} \triangleright (x\!:\!T_1 \to T_2)\rangle^l\ v$.
2. $\langle (x\!:\!T_1 \to T_2) \triangleright \{x\!:\!B \mid s\}\rangle^l\ v$.
3. $\langle \{x\!:\!B_1 \mid t_1\} \triangleright \{x\!:\!B_2 \mid t_2\}\rangle^l\ c$ *unless* $B_1 = B_2$ *and* $$t_2[x := c] \longrightarrow^* \texttt{true}$$

THEOREM 4 (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. $\square$

### 5.2 Type Correctness of Compilation

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

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

ASSUMPTION 5 (Soundness of $E \vdash^a_{alg} s \Rightarrow t$). *Suppose* $\vdash E$.

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

Note that this algorithm does not need to be complete (indeed, an extremely naive algorithm could simply return $E \vdash^?_{alg} s <: t$ 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 6 (Soundness of $E \vdash^a_{alg} S <: T$). *Suppose* $\vdash E$.

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

PROOF: By induction on derivations using Assumption 5. $\square$

Becasue algorithmic subtyping is sound, the hybrid compilation algorithm generates only well-typed programs:

LEMMA 7 (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 compilation derivations. $\square$

Since the generated code is well-typed, standard type-directed compilation and optimization techniques [39, 31] 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 (only when necessary).

**Figure 8: UpCast Insertion**

Upcast insertion $\boxed{E \vdash s \simeq t}$

$$\frac{}{E \vdash t \simeq t} \quad \text{[UP-REFL]}$$

$$\frac{E \vdash t_1 \simeq t_2 \qquad E \vdash t_2 \simeq t_3}{E \vdash t_1 \simeq t_3} \quad \text{[UP-TRANS]}$$

$$\frac{E \vdash S <: T}{E \vdash t \simeq \langle S \rhd T \rangle\, t} \quad \text{[UP-ADD]}$$

$$\frac{}{E \vdash t \simeq \lambda x{:}T.\, t\, x} \quad \text{[UP-ETA]}$$

$$\frac{}{E \vdash (\lambda x{:}S.\, t) \simeq (\lambda x{:}T.\, t)} \quad \text{[UP-FUNTY]}$$

$$\frac{E, x : T \vdash s \simeq t}{E \vdash (\lambda x{:}T.\, s) \simeq (\lambda x{:}T.\, t)} \quad \text{[UP-FUNBODY]}$$

$$\frac{E \vdash s \simeq s'}{E \vdash s\, t \simeq s'\, t} \quad \text{[UP-APPL]}$$

$$\frac{E \vdash t \simeq t'}{E \vdash s\, t \simeq s\, t'} \quad \text{[UP-APPR]}$$

$$\frac{E \vdash S = S'}{E \vdash \langle S \rhd T \rangle\, s \simeq \langle S' \rhd T \rangle^l\, s} \quad \text{[UP-CASTL]}$$

$$\frac{E \vdash T = T'}{E \vdash \langle S \rhd T \rangle\, s \simeq \langle S \rhd T' \rangle^l\, s} \quad \text{[UP-CASTR]}$$

$$\frac{E \vdash s \simeq t}{E \vdash \langle S \rhd T \rangle\, s \simeq \langle S \rhd T \rangle^l\, t} \quad \text{[UP-CASTBODY]}$$

### 5.3 Bisimulation

In this section we prove that compilation does not change the meaning of well-typed programs, so that the original and compiled programs are behaviorally equivalent, or *bisimilar*.

As a first step towards defining this bisimulation relation, the *cast insertion relation* $\simeq$ shown in Figure 8 characterizes some aspects of the relationship between source and compiled terms. The rule [UP-ADDCAST] states that, if $E \vdash S <: T$, then the cast $\langle S \rhd T \rangle$ is redundant, and any term $t$ is considered to be $\simeq$-equivalent to $\langle S \rhd T \rangle\, t$. Note that this rule requires that we track the current environment. The remaining rules implement the reflexive-transitive-compatible closure of this rule, updating the current environment as appropriate. The rule [UP-ETA] also allows for $\eta$-expansion, which is in part performed by the evaluation rule [E-CAST-FN] for function casts.

As a technical requirement, we assume that application of primitive functions preserves $\simeq$-equivalence:

ASSUMPTION 8 (Constant Bisimulation). *For all primitive functions c, if $s \simeq t$ then $[\![c]\!](s) \simeq [\![c]\!](t)$.*

The desired bisimulation relation $R$ is then obtained by strengthening the cast insertion relation with the additional requirement that both the original program and compiled programs are well-typed:

$$R = \{(s,t) \mid s \simeq t \text{ and } \exists S.\ \emptyset \vdash s : S \text{ and } \exists T.\ \emptyset \vdash t : T\}$$

This relation $R$ is a bisimulation relation, *i.e.*, if $R(s,t)$ holds then $s$ and $t$ exhibit equivalent behavior.

LEMMA 9 (Bisimulation). *Suppose $R(s,t)$.*

1. *If $s \longrightarrow s'$ then $\exists t'$ such that $t \longrightarrow^* t'$ and $R(s',t')$.*
2. *If $t \longrightarrow t'$ then $\exists s'$ such that $s \longrightarrow^* s'$ and $R(s',t')$.*

PROOF: By induction on the cast insertion derivation. □

Finally, we prove that the compilation $E \vdash s \hookrightarrow t : T$ of a well-typed program $s$ yields a bisimilar program $t$. Proving this property requires an inductive hypothesis that also characterizes the compilation relations $E \vdash s \hookrightarrow t \downarrow T$ and $E \vdash S \hookrightarrow T$.

LEMMA 10 (Compilation is Upcasting). *Suppose $\vdash E$.*

1. *If $E \vdash s : S$ and $E \vdash s \hookrightarrow t : T$ then $E \vdash T <: S$ and $E \vdash s \simeq t$.*
2. *If $E \vdash s : S$ and $E \vdash s \hookrightarrow t \downarrow T$ and $E \vdash S <: T$ then $E \vdash s \simeq t$.*
3. *If $E \vdash S$ and $E \vdash S \hookrightarrow T$ then $E \vdash S = T$.*

PROOF: By induction on compilation derivations. □

It follows that compilation does not change the behavior of well-typed programs.

LEMMA 11 (Correctness of Compilation). *Suppose $\emptyset \vdash s : S$ and $\emptyset \vdash s \hookrightarrow t : T$.*

1. *If $s \longrightarrow^* s'$ then $\exists t'$ such that $t \longrightarrow^* t'$ and $s' \simeq t'$.*
2. *If $t \longrightarrow^* t'$ then $\exists s'$ such that $s \longrightarrow^* s'$ and $s' \simeq t'$.*

PROOF: By Lemma 10, $s \simeq t$. Also, $t$ is well-typed by Lemma 7, and $s$ is also well-typed, so $R(s,t)$. The first case then follows by induction on the length of the reduction sequence $s \longrightarrow^* s'$. The base case clearly holds. For the inductive case, if $s \longrightarrow s'$ then by Lemma 9 $\exists t'$ such that $t \longrightarrow^* t'$ and $s' \simeq t'$. Furthermore, by the Preservation Theorem, $s'$ and $t'$ are well-typed, and so $R(s,t)$. The second case is similar. □

From part 3 of Lemma 10, compilation of a well-formed type yields an equivalent type. It follows via a straightforward induction that the compilation algorithm is guaranteed to accept all well-typed programs.

LEMMA 12 (Compilation Completeness). *Suppose $\vdash E$.*

1. *If $E \vdash s : S$ then $\exists t, T$ such that $E \vdash s \hookrightarrow t : T$.*
2. *If $E \vdash s : S$ and $E \vdash S <: T$ then $\exists t$ such that $E \vdash s \hookrightarrow t \downarrow T$.*
3. *If $E \vdash S$ then $\exists T$ such that $E \vdash S \hookrightarrow T$.*

PROOF: By induction on typing derivations. □

## 6. 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 conventional static type checkers.

To study this question, we assume the static type checker targets a restricted subset of $\lambda^H$ for which type checking is statically decidable. Specifically, we assume there exists a subset $\mathcal{D}$ of *Term* such that for all $t_1, t_2 \in \mathcal{D}$ and for all environments $E$ (containing only $\mathcal{D}$-terms), the judgment $E \vdash t_1 \Rightarrow t_2$ is decidable. We introduce the language $\lambda^S$ that is obtained from $\lambda^H$ by only permitting $\mathcal{D}$-terms in refinement types.

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

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

However, to yield a more general argument, we assume only that $\mathcal{D}$ is a subset of *Term* for which implication is decidable. It then follows that subtyping and type checking for $\lambda^S$ are also decidable, and we denote this type checking judgment as $E \vdash^{\mathsf{s}} t \,:\, T$.

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 $E$, the judgment $E \vdash^a_{alg} t_1 \Rightarrow t_2$ 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) $\lambda^S$ programs.

THEOREM 13. *For all $\lambda^S$ terms $t$, $\lambda^S$ environments $E$, and $\lambda^S$ 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 $\lambda^S$ types. The proof then follows by induction on typing derivations. $\square$

Thus, to a $\lambda^S$ 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 $\lambda^H$ programmer. To enable this comparison, we need to map expressive $\lambda^H$ types into the more restrictive $\lambda^S$ types, and in particular to map arbitrary boolean terms into $\mathcal{D}$-terms. We assume the computable function

$$\gamma : \textit{Term} \to \mathcal{D}$$

performs this mapping. The function *erase* then maps $\lambda^H$ refinement types to $\lambda^S$ refinement types by using $\gamma$ to abstract boolean terms:

$$\textit{erase}\{x\!:\!B \mid t\} \;=\; \{x\!:\!B \mid \gamma(t)\}$$

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

As might be expected, the *erase* function must lose information, with the consequence that for any computable mapping $\gamma$ there exists some program $P$ such that hybrid type checking of $P$ performs better than static type checking of *erase*$(P)$. In other words, because the hybrid type checker supports more precise specifications, it performs better than a traditional static type checker, which necessarily must work with less precise but decidable specifications.

THEOREM 14. *For any computable mapping $\gamma$ either:*

1. *the static type checker rejects the erased version of some well-typed $\lambda^H$ program, or*
2. *the static type checker accepts the erased version of some ill-typed $\lambda^H$ 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.*:

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

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:
$$E \vdash w \Rightarrow \texttt{false}$$
$$E \not\vdash \gamma(w) \Rightarrow \gamma(\texttt{false})$$
We construct as a counter-example the program $P_1$:
$$P_1 \;=\; \lambda x\!:\!\{x\!:\!\texttt{Int} \mid w\}.\,(x \;\texttt{as}\; \{x\!:\!\texttt{Int} \mid \texttt{false}\})$$
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 12 accepted by the hybrid type checker.
$$\emptyset \vdash P_1 \,:\, \{x\!:\!\texttt{Int} \mid w\} \to \{x\!:\!\texttt{Int} \mid \texttt{false}\}$$
However, from the assumption $E \not\vdash \gamma(w) \Rightarrow \gamma(\texttt{false})$ the erased version of the subtyping judgment does not hold:
$$\emptyset \not\vdash \textit{erase}(\{x\!:\!\texttt{Int} \mid w\}) <: \textit{erase}(\{x\!:\!\texttt{Int} \mid \texttt{false}\})$$
Hence *erase*$(P_1)$ is ill-typed and rejected by the static type checker.
$$\forall T.\;\; \emptyset \not\vdash^{\mathsf{s}} \textit{erase}(P_1) \,:\, T$$
2. Conversely, suppose:
$$E \not\vdash w \Rightarrow \texttt{false}$$
$$E \vdash \gamma(w) \Rightarrow \gamma(\texttt{false})$$
From the first supposition and by the definition of the implication judgment, there exists integers $n$ and $m$ such that
$$w[x := n] \longrightarrow^m \texttt{true}$$
We now construct as a counter-example the program $P_2$:
$$P_2 \;=\; \lambda x\!:\!\{x\!:\!\texttt{Int} \mid w\}.\,(x \;\texttt{as}\; \{x\!:\!\texttt{Int} \mid \texttt{false} \wedge (n = m)\})$$
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`.
$$\frac{t[x := a] \longrightarrow^b \texttt{true}}{E \vdash^{\times}_{alg} t \Rightarrow (\texttt{false} \wedge a = b)}$$
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:
$$\vdash^{\times}_{alg} \{x\!:\!\texttt{Int} \mid w\} <: \{x\!:\!\texttt{Int} \mid \texttt{false} \wedge (n = m)\}$$
Therefore, the hybrid type checker rejects $P_2$, which by Lemma 12 is therefore ill-typed.
$$\forall P, T.\;\; \not\vdash P_2 \hookrightarrow P \,:\, T$$
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

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

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

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

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

$$P_3 = \lambda x : \{x : \texttt{Int} \mid \texttt{false}\}. \ (x \ \texttt{as} \ \{x : \texttt{Int} \mid \texttt{false} \wedge (n{=}m)\})$$

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

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

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

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

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

$$\forall T. \ \ \emptyset \not\vdash^{S} \mathit{erase}(P_3) \ : \ T$$

□

## 7. Related Work

Much prior work has focused on dynamic checking of expressive specifications, or *contracts* [30, 14, 26, 19, 24, 27, 36, 25]. 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 [30] 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 [1] 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 [22]. Thatte developed a similar system in which the necessary casts are implicit [38]. 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 [17] 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 [45, 44]. 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.

In recent work, Ou, Tan, Mandelbaum, and Walker developed a type system similar to ours that combines dynamic checks with refinement and dependent types [35]. They leverage dynamic checks to reduce the need for precise type annotations in explicitly labeled regions of programs. Unlike our approach, their type system is decidable, since they do not support arbitrary refinement predicates. Their system can also handle mutable data.

The static checking tool ESC/Java [16] checks expressive JML specifications [8, 26] using the Simplify automatic theorem prover [12]. 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 [33] 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 [41, 34, 2].

Prior work (*e.g.* [7]) introduced and studied implicit coercions in type systems. Note that there are no implicit coercions in the $\lambda^H$ type system itself, but only in the compilation algorithm, and so we do not need a coherence theorem for $\lambda^H$, but instead reason about the connection between the type system and compilation algorithm.

## 8. Conclusions and Future Work

Precise specifications are essential for modular software development. Hybrid type checking appears to be a promising approach for providing high coverage checking of precise specifications. This paper explores hybrid type checking in the idealized context of the $\lambda$-calculus, and highlights some of the key principles and implications of hybrid type checking.

Many areas remain to be explored, such as how hybrid type checking interacts with the many features of realistic programming languages, such as records, variants, recursive types, polymorphism, type operators, side-effects, exceptions, objects, concurrency, etc. Our initial investigations suggests that hybrid type checking can be extended to support these additional features, though with some restrictions. In an imperative context, we might require that refinement predicates be `pure` [45].

Another important area of investigation is type inference for hybrid type systems. A promising approach is to develop type inference algorithms that infer most type annotations, and to use occasional dynamic checks to compensate for limitations in the type inference algorithm.

In terms of software deployment, an important topic is recovery methods for post-deployment cast failures; transactional roll-back mechanisms [21, 40] 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 [32] and typed assembly language [31].

**Acknowledgements** Thanks to Matthias Felleisen, Stephen Freund, Robby Findler, Martín Abadi, Shriram Krishnamurthi, David Walker, Aaron Tomb, Kenneth Knowles, and Jessica Gronski for valuable feedback on this paper. This work was supported by the National Science Foundation under Grants CCR-0341179, and by faculty research funds granted by the University of California at Santa Cruz.

## References


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

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

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

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

[5] T. Ball, R. Majumdar, T. Millstein, and S. Rajamani. Predicate abstraction of C programs. In *Proceedings of the Conference on Programming Language Design and Implementation*, pages 203–213, June 2001.

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

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

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

[9] L. Cardelli. Phase distinctions in type theory. Manuscript, 1988.

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

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

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

[13] R. B. Findler. *Behavioral Software Contracts*. PhD thesis, Rice University, 2002.

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

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

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

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

[18] P. Godefroid, N. Klarlund, and K. Sen. DART: Directed automated random testing. In *Proceedings of the ACM Conference on Programming Language Design and Implementation*, pages 213–223, 2005.

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

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

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

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

[23] T. A. Henzinger, R. Jhala, R. Majumdar, G. C. Necula, G. Sutre, and W. Weimer. Temporal-safety proofs for systems code. In *Proceedings of the IEEE Conference on Computer Aided Verification*, pages 526–538, 2002.

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

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

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

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

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

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

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

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

[32] G. C. Necula. Proof-carrying code. In *Proceedings of the ACM Symposium on Principles of Programming Languages*, pages 106–119, 1997.

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

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

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

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

[37] Reynolds, J.C. Definitional interpreters for higher-order programming languages. In *Proc. ACM Annual Conference*, pages 717–740, 1972.

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

[39] D. Tarditi, G. Morrisett, P. Cheng, C. Stone, R. Harper, and P. Lee. TIL: A type-directed optimizing compiler for ML. *ACM SIGPLAN Notices*, 31(5):181–192, 1996.

[40] J. Vitek, S. Jagannathan, A. Welc, and A. L. Hosking. A semantic framework for designer transactions. In *Proceedings of European Symposium on Programming*, pages 249–263, 2004.

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

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

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

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

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

## A. Proofs from Subsection 5.1

LEMMA 15 (Weakening). *Suppose*

$$E = E_1, E_2$$
$$E' = E_1, x : P, E_2$$

*Then:*

1. *If* $E' \models (\sigma, x := t)$ *then* $E \models \sigma$.
2. *If* $E \vdash s \Rightarrow t$ *then* $E' \vdash s \Rightarrow t$.
3. *If* $E \vdash S <: T$ *then* $E' \vdash S <: T$.
4. *If* $E \vdash t : T$ *then* $E' \vdash t : T$.
5. *If* $E \vdash T$ *then* $E' \vdash T$.

PROOF: By simultaneous induction on typing derivations. □

LEMMA 16 (Narrowing). *Suppose*

$$E_1 \vdash P <: Q$$
$$E = E_1, x : Q, E_2$$
$$E' = E_1, x : P, E_2$$

*Then:*

1. *If* $E' \models \sigma$ *then* $E \models \sigma$.
2. *If* $E \vdash s \Rightarrow t$ *then* $E' \vdash s \Rightarrow t$.
3. *If* $E \vdash S <: T$ *then* $E' \vdash S <: T$.
4. *If* $E \vdash t : T$ *then* $E' \vdash t : T$.
5. *If* $E \vdash T$ *then* $E' \vdash T$.

PROOF: By simultaneous induction on typing derivations. □

LEMMA 17. *Subtyping is reflexively-transitively closed:*

1. $E \vdash T <: T$.
2. *If* $E \vdash T_1 <: T_2$ *and* $E \vdash T_2 <: T_3$ *then* $E \vdash T_1 <: T_3$ .

PROOF: By induction on typing derivations. □

LEMMA 18 (Substitution). *Suppose*

$$E_1 \vdash s : S$$
$$\theta = (x := s)$$
$$E = E_1, x : S, E_2$$
$$E' = E_1, \theta E_2$$

*Then:*

1. *If* $E' \models (\sigma_1, \sigma_2)$ *where* $|E_1| = |\sigma_1|$ *then* $E \models (\sigma_1, x := \sigma_1(s), \sigma_2)$.
2. *If* $E \vdash t_1 \Rightarrow t_2$ *then* $E' \vdash \theta t_1 \Rightarrow \theta t_2$.
3. *If* $E \vdash T_1 <: T_2$ *then* $E' \vdash \theta T_1 <: \theta T_2$.
4. *If* $E \vdash T$ *then* $E' \vdash \theta T$.
5. *If* $E \vdash t : T$ *then* $E' \vdash \theta t : \theta T$.

PROOF: By simultaneous induction on typing derivations. □

LEMMA 19 (Canonical Forms). *If* $\vdash v : (x : T_1 \to T_2)$ *then either*

1. $v = \lambda x\!:\!S.\, s$ *and* $\vdash T_1 <: S$ *and* $x : S \vdash s : T_2$, *or*
2. $v$ *is a constant and ty*$(c)$ *is a subtype of* $x\!:\!T_1 \to T_2$.

## B. Proofs from Subsection 5.2

## C. Proofs from Subsection 5.3

LEMMA 20 (UpCast Substitution).
*If* $E, x : T \vdash s \simeq s'$ *and* $E \vdash t \simeq t'$ *then* $E \vdash s[x := t] \simeq s'[x := t']$.

PROOF: By induction on the derivation that $s \simeq s'$. □

LEMMA 21 (Inversion of the UpCast Relation).
*If* $\lambda x\!:\!S.\, s \simeq v$ *then* $v = \lambda x\!:\!T.\, t$ *and* $s \simeq t$.

PROOF: By induction on the derivation $\lambda x\!:\!S.\, s \simeq v$. □

LEMMA 22 (Bisimilar Values). *Suppose* $R(s,t)$ *and* $s$ *a value. Then* $\exists t'$ *such that* $t \longrightarrow^* t'$ *and* $R(s,t')$ *and* $t'$ *a value.*

PROOF: By induction on $s \simeq t$. All cases are straightforward except for [UP-ADDCAST], for which we perform a case analysis on the casted type. □

LEMMA 23. *If* $\vdash (E, x : U)$ *and* $E, x : U \vdash S$ *and* $E, x : U \vdash S <: T$ *and* $E \vdash s : U$ *and* $E \vdash s \simeq t$ *then* $E \vdash S[x := s] <: T[x := t]$.

PROOF: By induction on the subtyping derivation. □

RESTATEMENT OF LEMMA 9 (Bisimulation) *Suppose* $R(s,t)$.

1. *If* $s \longrightarrow s'$ *then* $\exists t'$ *such that* $t \longrightarrow^* t'$ *and* $R(s',t')$.
2. *If* $t \longrightarrow t'$ *then* $\exists s'$ *such that* $s \longrightarrow^* s'$ *and* $R(s',t')$.

PROOF: The proof of part 1 is by induction on derivation that $s \simeq t$.

- [UP-REFL] This case clearly holds.
- [UP-TRANS] Suppose $s \simeq t$ via $s \simeq u$ and $u \simeq t$. By induction $\exists u'$ such that $u \longrightarrow^* u'$ and $s' \simeq u'$. Therefore (by an additional induction argument of the length of $u \longrightarrow^* u'$) $\exists t'$ such that $t \longrightarrow^* t'$ and $u' \simeq t'$. Hence $s' \simeq t'$.
- [UP-ETA] Suppose $t = \lambda x : T.\, s\ x$. Take $t' = \lambda x : T.\, s'\ x$ and then $s' \simeq t'$ and $t \longrightarrow t'$.
- [UP-FUNTY], [UP-FUNBODY] Contradicts assumption that $s$ is reducible.
- [UP-ADD] Suppose $t = \langle S \rhd T \rangle\ s$. Take $t' = \langle S \rhd T \rangle\ s'$ and then $s' \simeq t'$ and $t \longrightarrow t'$.
- [UP-CASTL] Suppose $s = \langle S \rhd T \rangle\ s_1 \simeq \langle S' \rhd T \rangle\ s_1 = t$ where $\vdash S = S'$. We perform a cast analysis on $s \longrightarrow s'$.
  - [E-COMPAT] Suppose $s \longrightarrow \langle S \rhd T \rangle\ s_1'$ because $s_1 \longrightarrow s_1'$. Take $t' = \langle S' \rhd T \rangle\ s_1'$ and then $s' \simeq t'$ and $t \longrightarrow t'$.
  - [E-CAST-F] Suppose
    $$\begin{array}{rcl} s & = & \langle x\!:\!S_1 \to S_2 \rhd x\!:\!T_1 \to T_2 \rangle\ s_1 \\ \longrightarrow\quad s' & = & \lambda x\!:\!T_1.\, \langle S_2 \rhd T_2 \rangle^l\ (t\ (\langle T_1 \rhd S_1 \rangle^l\ x)) \end{array}$$
    where $S' = x : S_1' \to S_2'$ and $\vdash S_1 = S_1'$ and $\vdash S_2 = S_2'$. Take $t' = \lambda x\!:\!T_1.\, \langle S_2' \rhd T_2 \rangle^l\ (t\ (\langle T_1 \rhd S_1' \rangle^l\ x))$ and then $s' \simeq t'$ and $t \longrightarrow t'$.
  - [E-CAST-C] Suppose $s = \langle S \rhd \{x : B \mid t\} \rangle\ c \longrightarrow c = s'$ because $t[x := c] \longrightarrow^* \texttt{true}$. Take $t' = c$ and then $s' \simeq t'$ and $t \longrightarrow t'$.
- [UP-CASTR] Similar to [UP-CASTL].
- [UP-APPR] Suppose $s = s_1\ s_2 \simeq s_1\ t_2 = t$ because $s_2 \simeq t_2$. We perform a cast analysis on $s \longrightarrow s'$.
  - [E-COMPAT] Straightforward.
  - [E-$\beta$] Suppose $s = (\lambda x : S.\, s_3)\ s_2 \longrightarrow s' = s_3[x := s_2]$. Take $t' = s_3[x := t_2]$ and by Lemma 20 we have that $s' \simeq t'$ and $t \longrightarrow t'$.
  - [E-PRIM] Suppose $s = c\ s_2 \longrightarrow s' = [\![c]\!]\ s_2$. Take $t' = [\![c]\!]\ t_2$ and by Assumption 8 we have that $s' \simeq t'$ and $t \longrightarrow t'$.

- [UP-APPL] Suppose $s = s_1\ s_2 \simeq t_1\ s_2 = t$ because $s_1 \simeq t_1$. We perform a cast analysis on $s \longrightarrow s'$.
  - [E-COMPAT] Straightforward.
  - [E-$\beta$] Suppose $s = (\lambda x : S.\, s_3)\ s_2 \longrightarrow s' = s_3[x := s_2]$. By Lemma 21, $t_1 = \lambda x : T.\, t_3$ and $s_3 \simeq t_3$. Take $t' = t_3[x := s_2]$ and then by Lemma 20 we have that $s' \simeq t'$ and $t \longrightarrow t'$.
  - [E-PRIM] Suppose $s = c\ s_2 \longrightarrow s' = [\![c]\!]\ s_2$. We prove by induction on the derivation that $c \simeq t_1$ that there exists $t'$ such that $t_1\ s_2 \longrightarrow^* t' \simeq c\ s_2$.
    - [UP-REFL], [UP-TRANS] Straightforward.
    - [UP-ETA] Suppose $t_1 = \lambda x{:}T.\, cx$ then $t_1\ s_2 \longrightarrow c\ s_2$.
    - [UP-ADD] Suppose $t_1 = \langle S \rhd T\rangle\ c$. Since $s$ is well-typed, $c$ must have a function type, and since $t$ is well-typed $t_1$ must be a function cast. Hence

$$\begin{array}{lll} t & = & (\langle x{:}S_1 \to S_2 \rhd x{:}T_1 \to T_2\rangle\ c)\ s_2 \\ & \longrightarrow & (\lambda x{:}T_1.\, \langle S_2 \rhd T_2\rangle^l\, (c\ (\langle T_1 \rhd S_1\rangle^l\, x)))\ s_2 \\ & \longrightarrow & \langle S_2 \rhd T_2\rangle^l\, (c\ (\langle T_1 \rhd S_1\rangle^l\, s_2)) \\ & = & t' \simeq c\ s_2 \end{array}$$

    - No other rules are applicable.
- [UP-CASTBODY] Suppose $s = \langle S \rhd T\rangle\ s_1 \simeq \langle S \rhd T\rangle\ t_1 = t$ because $s_1 \simeq t_1$. We perform a cast analysis on $s \longrightarrow s'$.
  - [E-COMPAT] Straightforward.
  - [E-CAST-F] Suppose

$$\begin{array}{lll} s & = & \langle x{:}S_1 \to S_2 \rhd x{:}T_1 \to T_2\rangle\ s_1 \\ & \longrightarrow & \lambda x{:}T_1.\, \langle S_2 \rhd T_2\rangle^l\, (s_1\ (\langle T_1 \rhd S_1\rangle^l\, x)) \end{array}$$

    Take $t' = \lambda x{:}T_1.\, \langle S_2 \rhd T_2\rangle^l\, (t_1\ (\langle T_1 \rhd S_1\rangle^l\, x))$ and then $s' \simeq t'$ and $t \longrightarrow t'$.
  - [E-CAST-C] Suppose $s = \langle S \rhd \{x : B \mid t\}\rangle\ c \longrightarrow c = s'$ because $t[x := c] \longrightarrow^* \texttt{true}$, where $s_1 = c$. By this bisimulation lemma, $t[x := t_1] \longrightarrow^* \texttt{true}$, and so $s = \langle S \rhd \{x{:}B \mid t\}\rangle\ t_1 \longrightarrow t_1$. Take $t' = t_1$ and then $s' \simeq t'$ and $t \longrightarrow t'$.

The proof of part 2 of Lemma 9 is again by induction on derivation that $s \simeq t$.

- [UP-REFL] This case clearly holds.
- [UP-TRANS] Suppose $s \simeq t$ via $s \simeq u$ and $u \simeq t$. By induction $\exists u'$ such that $u \longrightarrow^* u'$ and $u' \simeq t'$. Therefore (by an additional induction argument of the length of $u \longrightarrow^* u'$) $\exists s'$ such that $s \longrightarrow^* s'$ and $s' \simeq u'$. Hence $s' \simeq t'$.
- [UP-ETA], [UP-FUNTY], [UP-FUNBODY] Contradicts assumption that $t$ is reducible.
- [UP-ADD] Suppose $t = \langle S \rhd T\rangle\ s_1$ where $\vdash S <: T$. We perform a cast analysis on $t \longrightarrow t'$.
  - [E-COMPAT] Straightforward.
  - [E-CAST-F] Suppose

$$\begin{array}{llll} & t & = & \langle x{:}S_1 \to S_2 \rhd x{:}T_1 \to T_2\rangle\ s_1 \\ \longrightarrow & t' & = & \lambda x{:}T_1.\, \langle S_2 \rhd T_2\rangle^l\, (s_1\ (\langle T_1 \rhd S_1\rangle^l\, x)) \end{array}$$

    where $\vdash T_1 <: S_1$ and $x : T_1 \vdash S_2 <: T_2$. Take $s' = s$ and then $s' \simeq t'$.
  - [E-CAST-C] Suppose $t = \langle S \rhd \{x : B \mid t_1\}\rangle\ c \longrightarrow c = t'$. Then $s = c$. Take $s' = s$.
- [UP-APPL] Suppose $s = s_1\ s_2 \simeq t_1\ s_2 = t$ because $s_1 \simeq t_1$. We perform a cast analysis on $t \longrightarrow t'$.
  - [E-COMPAT] Straightforward.
  - [E-$\beta$] Suppose $t = (\lambda x : S.\, t_3)\ s_2 \longrightarrow t' = t_3[x := s_2]$. We prove by induction on the derivation that $s_1 \simeq t_1 = lamxSt_3$ that there exists $s'$ such that $s_1\ s_2 \longrightarrow^* s' \simeq t'$.
    - [UP-REFL], [UP-TRANS], [UP-FUNTY], [UP-FUNBODY] Straightforward.
    - [UP-ETA] Then $t_3 = s_1\ x$. Take $s' = s$ and then $s' \simeq t'$.
    - No other rules are applicable.
  - E-Prim Suppose $t = c\ s_2 \longrightarrow t' = [\![c]\!]\ s_2$. Then $s_1 = t_1 = c$ and this case clearly holds.
- [UP-APPR] Suppose $s = s_1\ s_2 \simeq s_1\ t_2 = t$ because $s_2 \simeq t_2$. We perform a cast analysis on $t \longrightarrow t'$.
  - [E-COMPAT] Straightforward.
  - [E-$\beta$] Suppose $t = (\lambda x : S.\, s_3)\ t_2 \longrightarrow t' = s_3[x := t_2]$. Take $s' = s_3[x := s_2]$ and by Lemma 20 we have that $s' \simeq t'$ and $s \longrightarrow s'$.
  - [E-PRIM] Suppose $t = c\ t_2 \longrightarrow t' = [\![c]\!]\ t_2$. Take $s' = [\![c]\!]\ s_2$ and by Assumption 8 we have that $s' \simeq t'$ and $s \longrightarrow s'$.
- [UP-CASTL] Suppose $s = \langle S \rhd T\rangle\ s_1 \simeq \langle S' \rhd T\rangle\ s_1 = t$ where $\vdash S = S'$. We perform a cast analysis on $s \longrightarrow s'$.
  - [E-COMPAT] Straightforward.
  - [E-CAST-F] Suppose

$$\begin{array}{llll} & t & = & \langle x{:}S_1' \to S_2' \rhd x{:}T_1 \to T_2\rangle\ s_1 \\ \longrightarrow & t' & = & \lambda x{:}T_1.\, \langle S_2' \rhd T_2\rangle^l\, (t\ (\langle T_1 \rhd S_1'\rangle^l\, x)) \end{array}$$

    where $S' = x : S_1' \to S_2'$ and $S = x : S_1 \to S_2$ and $\vdash S_1 = S_1'$ and $\vdash S_2 = S_2'$. Take $s' = \lambda x{:}T_1.\, \langle S_2 \rhd T_2\rangle^l\, (t\ (\langle T_1 \rhd S_1\rangle^l\, x))$ and then $s' \simeq t'$ and $s \longrightarrow s'$.
  - [E-CAST-C] Suppose $t = \langle S \rhd \{x : B \mid t_1\}\rangle\ c \longrightarrow c = t'$ because $t_1[x := c] \longrightarrow^* \texttt{true}$. Take $s' = c$ and then $s' \simeq t'$ and $s \longrightarrow s'$.
- [UP-CASTR] Similar to [UP-CASTL].
- [UP-CASTBODY] Suppose $s = \langle S \rhd T\rangle\ s_1 \simeq \langle S \rhd T\rangle\ t_1 = t$ because $s_1 \simeq t_1$. We perform a cast analysis on $s \longrightarrow s'$.
  - [E-COMPAT] Straightforward.
  - [E-CAST-F] Suppose

$$\begin{array}{lll} t & = & \langle x{:}S_1 \to S_2 \rhd x{:}T_1 \to T_2\rangle\ t_1 \\ & \longrightarrow & \lambda x{:}T_1.\, \langle S_2 \rhd T_2\rangle^l\, (t_1\ (\langle T_1 \rhd S_1\rangle^l\, x)) \end{array}$$

    Take $s' = \lambda x{:}T_1.\, \langle S_2 \rhd T_2\rangle^l\, (s_1\ (\langle T_1 \rhd S_1\rangle^l\, x))$ and then $s' \simeq t'$ and $s \longrightarrow s'$.
  - [E-CAST-C] Suppose $t = \langle S \rhd \{x : B \mid t_2\}\rangle\ c \longrightarrow c = t'$ because $t_2[x := c] \longrightarrow^* \texttt{true}$, where $t_1 = c$. Then $s_1 = c$. Take $s' = c$ and then $s' \simeq t'$ and $s \longrightarrow s'$.

□

RESTATEMENT OF LEMMA 10 (Compilation is Upcasting) *Suppose* $\vdash E$.

1. *If* $E \vdash s : S$ *and* $E \vdash s \hookrightarrow t : T$ *then* $E \vdash T <: S$ *and* $E \vdash s \simeq t$.
2. *If* $E \vdash s : S$ *and* $E \vdash s \hookrightarrow t \downarrow T$ *and* $E \vdash S <: T$ *then* $E \vdash s \simeq t$.

*3. If $E \vdash S$ and $E \vdash S \hookrightarrow T$ then $E \vdash S = T$.*

PROOF: By simultaneous induction on compilation derivations.
Part 1.

- [C-VAR], [C-CONST]. Straightforward.
- [C-FUN] In this case

$$\begin{array}{ll} E \vdash (\lambda x\!:\!S.\, s) \hookrightarrow (\lambda x\!:\!T.\, t) : (x\!:\!T \to T') & \\ E \vdash S \hookrightarrow T & \text{antecedent} \\ E, x : T \vdash s \hookrightarrow t : T' & \text{antecedent} \\ & \\ E \vdash (\lambda x\!:\!S.\, s) : (x\!:\!S \to S') & \\ E \vdash S & \text{antecedent} \\ E, x : S \vdash s : S' & \text{antecedent} \\ & \\ E \vdash S = T & \text{induction} \\ E, x : T \vdash s : S' & \text{Lemma 18} \\ E, x : T \vdash s \simeq t & \text{induction} \\ E \vdash \lambda x\!:\!T.\, s \simeq \lambda x\!:\!T.\, t & [\text{UP-FUNBODY}] \\ E \vdash \lambda x\!:\!S.\, s \simeq \lambda x\!:\!T.\, t & [\text{UP-FUNTY}] \\ & \\ E, x : T \vdash T' <: S' & \text{induction} \\ E \vdash (x\!:\!T \to T') <: (x\!:\!S \to S') & [\text{S-ARROW}] \end{array}$$

- [C-APP] In this case

$$\begin{array}{ll} E \vdash (s_1\ s_2) \hookrightarrow (t_1\ t_2) : T'[x := t_2] & \\ E \vdash s_1 \hookrightarrow t_1 : (x\!:\!T \to T') & \text{antecedent} \\ E \vdash s_2 \hookrightarrow t_2 \downarrow T & \text{antecedent} \\ & \\ E \vdash s_1\ s_2 : S[x := s_2] & \\ E \vdash s_1 : (x\!:\!S \to S') & \text{antecedent} \\ E \vdash s_2 : S & \text{antecedent} \\ & \\ E \vdash s_1 \simeq t_1 & \text{induction} \\ E \vdash (x\!:\!T \to T') <: (x\!:\!S \to S') & \text{induction} \\ E \vdash S <: T & [\text{S-ARROW}] \\ E \vdash s_2 \simeq t_2 & \text{induction} \\ E \vdash s_1\ s_2 \simeq t_1\ t_2 & [\text{UP-}\cdots] \\ E, x : S \vdash T' <: S' & [\text{S-ARROW}] \\ E, x : S \vdash T'[x := t_2] <: S'[x := s_2] & \text{Lemma 23} \end{array}$$

- [C-CAST] In this case

$$\begin{array}{ll} E \vdash \langle S_1 \rhd S_2 \rangle\, s \hookrightarrow \langle T_1 \rhd T_2 \rangle\, t : T_2 & \\ E \vdash S_1 \hookrightarrow T_1 & \text{antecedent} \\ E \vdash S_2 \hookrightarrow T_2 & \text{antecedent} \\ E \vdash s \hookrightarrow t \downarrow^l T_1 & \text{antecedent} \\ & \\ E \vdash \langle S_1 \rhd S_2 \rangle\, s : S & \\ E \vdash s : S_1 & \text{antecedent} \\ E \vdash S_2 : & \text{antecedent} \\ & \\ E \vdash S_1 = T_1 & \text{induction} \\ E \vdash S_2 = T_2 & \text{induction} \\ E \vdash s \simeq t & \text{induction} \\ E \vdash \langle S_1 \rhd S_2 \rangle\, s \simeq \langle T_1 \rhd T_2 \rangle\, t & [\text{UP-}\cdots] \end{array}$$

Part 2.

- [CC-OK] In this case

$$\begin{array}{ll} E \vdash s \hookrightarrow t \downarrow^T & \\ E \vdash s \hookrightarrow t : S' & \text{antecedent} \\ E \vdash s : S & \\ E \vdash s \simeq t & \text{induction} \end{array}$$

- [CC-CHK] In this case

$$\begin{array}{ll} E \vdash s \hookrightarrow t \downarrow^T \ E \vdash s \hookrightarrow t : S' & \text{antecedent} \\ & \\ E \vdash s : S & \\ E \vdash S <: T & \\ E \vdash s \simeq t & \text{induction} \\ E \vdash s \simeq \langle S \rhd T \rangle\, t & [\text{UP-ADD}] \end{array}$$

Part 3.

- [C-BASE] Suppose $E \vdash \{x : B \,|\, s\} \hookrightarrow \{x : B \,|\, t\}$ via $E, x : B \vdash s \hookrightarrow t : \{y : \texttt{Bool} \,|\, t'\}$. By [WT-BASE] $E, x : B \vdash s : \texttt{Bool}$. By Lemma 7, $E, x : B \vdash t : \texttt{Bool}$.

  Let $\sigma$ be any substitution consistent with $E, x : B$. Then by Lemma 20 and 18, we have that

$$\begin{array}{l} \vdash \sigma(s) \hookrightarrow \sigma(t) : (\{y\!:\!\texttt{Bool} \,|\, \sigma(t')\}) \\ \vdash \sigma(s) : \texttt{Bool} \\ \vdash \sigma(t) : \texttt{Bool} \end{array}$$

  Hence $R(\sigma(s), \sigma(t))$, and so by Lemma 9, $\sigma(s) \longrightarrow^* \texttt{true}$ iff $\sigma(t) \longrightarrow^* \texttt{true}$. Thus

$$\begin{array}{l} E, x : B \vdash s \Rightarrow t \\ E, x : B \vdash t \Rightarrow s \end{array}$$

  Hence $E \vdash \{x\!:\!B \,|\, s\} = \{x\!:\!B \,|\, t\}$.

- [C-ARROW] In this case

$$\begin{array}{ll} E \vdash (x\!:\!S_1 \to S_2) \hookrightarrow (x\!:\!T_1 \to T_2) & \\ E \vdash S_1 \hookrightarrow T_1 & \text{antecedent} \\ E, x : T_1 \vdash S_2 \hookrightarrow S_2 & \text{antecedent} \\ & \\ E \vdash (x\!:\!S_1 \to S_2) & \\ E \vdash S_1 & \text{antecedent} \\ E, x : S_1 \vdash S_2 & \text{antecedent} \\ & \\ E \vdash S_1 = T_1 & \text{induction} \\ E, x : T_1 \vdash S_2 & \text{Lemma 18} \\ E, x : T_1 \vdash S_2 = T_2 & \text{induction} \\ E \vdash (x\!:\!S_1 \to S_2) <: (x\!:\!T_1 \to T_2) & [\text{S-ARROW}] \\ & \\ E, x : S_1 \vdash S_2 = T_2 & \text{Lemma 18} \\ E \vdash (x\!:\!T_1 \to T_2) <: (x\!:\!S_1 \to S_2) & [\text{S-ARROW}] \end{array}$$

□

RESTATEMENT OF LEMMA 12 (Compilation Completeness) *Suppose $\vdash E$.*

1. *If $E \vdash s : S$ then $\exists t, T$ such that $E \vdash s \hookrightarrow t : T$.*
2. *If $E \vdash s : S$ and $E \vdash S <: T$ then $\exists t$ such that $E \vdash s \hookrightarrow t \downarrow T$.*
3. *If $E \vdash S$ then $\exists T$ such that $E \vdash S \hookrightarrow T$.*

PROOF: By induction on the typing derivation.

1. - [T-VAR], [T-CONST] Straightforward.
   - [T-FUN] In this case, there exists $T, t, T'$ such that

$$\begin{array}{ll} E \vdash (\lambda x\!:\!S.\, s) : (x\!:\!S \to S') & \text{conclusion} \\ E \vdash S & \text{antecedent} \\ E, x : S \vdash s : S' & \text{antecedent} \\ & \\ E \vdash S \hookrightarrow T & \text{induction} \\ E \vdash S = T & \text{Lemma 10} \\ E, x : T \vdash s : S' & \text{Lemma 18} \\ E \vdash s \hookrightarrow t : T' & \text{induction} \\ E \vdash (\lambda x\!:\!S.\, s) \hookrightarrow (\lambda x\!:\!T.\, t) : (x\!:\!T \to T') & [\text{C-FUN}] \end{array}$$

- [T-APP] In this case, there exists $T_1, t, T, T'$ such that

| | |
|---|---|
| $E \vdash s_1\ s_2 \ :\ S'[x := s_2]$ | conclusion |
| $E \vdash t_1 \ :\ (x{:}S \rightarrow S')$ | antecedent |
| $E \vdash t_2 \ :\ S$ | antecedent |
| | |
| $E \vdash s_1 \ \hookrightarrow\ t_1 \ :\ T_1$ | induction |
| $E \vdash T_1 <: (x{:}S \rightarrow S')$ | Lemma 10 |
| $T_1 = (x{:}T \rightarrow T')$ | [S-ARROW] |
| $E \vdash S <: T$ | [S-ARROW] |
| $E \vdash s_2 \ \hookrightarrow\ t_2 \downarrow T$ | induction |
| $E \vdash s_1\ s_2 \ \hookrightarrow\ t_1\ t_2 \ :\ T'[x := t_2]$ | [C-APP] |

- [T-CAST] In this case, there exists $T_1, T_2, t$ such that

| | |
|---|---|
| $E \vdash \langle S_1 \rhd S_2 \rangle\ s \ :\ S_2$ | conclusion |
| $E \vdash s \ :\ S_1$ | antecedent |
| $E \vdash S_2$ | antecedent |
| | |
| $E \vdash S_1$ | |
| $E \vdash S_1 \ \hookrightarrow\ T_1$ | induction |
| $E \vdash S_2 \ \hookrightarrow\ T_2$ | induction |
| $E \vdash S_1 = T_1$ | Lemma 10 |
| $E \vdash s \hookrightarrow t \downarrow T_1$ | induction |
| $E \vdash \langle S_1 \rhd S_2 \rangle\ s \ \hookrightarrow\ \langle T_1 \rhd T_2 \rangle\ s \ :\ T_2$ | [C-CAST] |

- [T-SUB] If $E \vdash s \ :\ S$ via [T-SUB] then $E \vdash s \ :\ S'$ for some $S'$ and this case holds by induction

2. From $E \vdash s \ :\ S$ by induction there exists $t, U$ such that $E \vdash s \ \hookrightarrow\ t \ :\ U$. By Lemma 10, $E \vdash U <: S$, and by the transitivity of subtyping, $E \vdash U <: T$.

   If $E \vdash^{\surd}_{alg} U <: T$ then $E \vdash s \ \hookrightarrow\ t \downarrow T$ via [CC-OK].

   Otherwise, by Lemma 6, $E \vdash^{?}_{alg} U <: T$, and hence $E \vdash s \ \hookrightarrow\ \langle U \rhd T \rangle\ t \downarrow T$ via [CC-CHK].

3. [WT-ARROW], [WT-BASE] Straightforward.

□