Sage: Unified Hybrid Checking for First-Class Types, General Refinement Types, and `Dynamic` (Extended Report)

Kenneth Knowles[†] Aaron Tomb[†] Jessica Gronski[†] Stephen N. Freund[‡] Cormac Flanagan[†]

[†]University of California, Santa Cruz [‡] Williams College

Abstract

This paper presents Sage, a functional programming language with a rich type system that supports a broad range of typing paradigms, from dynamically-typed Scheme-like programming, to decidable ML-like types, to precise refinement types. This type system is a synthesis of three general concepts — first-class types, general refinement types, and the type `Dynamic` — that add expressive power in orthogonal and complementary ways.

None of these concepts are statically decidable. The Sage compiler uniformly circumvents this limitation using hybrid type checking, which inserts occasional run-time casts in particularly complicated situations that cannot be statically checked. We describe a prototype implementation of Sage and preliminary experimental results showing that most or all types are enforced via static type checking — the number of compiler-inserted casts is very small or zero on all our benchmarks.

1. Introduction

The design of an advanced type system typically involves a difficult trade-off between expressiveness, complexity, and static decidability. This paper describes the Sage programming language and type system, which explores an unusual yet rather promising point in this design space. Sage is a purely functional programming language with a minimalist design philosophy. It extends the three constructs of the lambda-calculus (abstraction, application, and variable reference) with only three additional constructs. Yet within this minimal syntactic core, Sage provides a sophisticated type system that is quite expressive and flexible.

This combination of simplicity and power is achieved by a synthesis of the following general concepts, each of which extends the expressive power of the type system in orthogonal and complementary ways. The expressive power of these concepts means that none are statically decidable. Sage uniformly circumvents this limitation via hybrid type checking, described below.

First-Class Types. Sage eschews the term/type/kind hierarchy common in type systems and instead *unifies* the syntactic categories of terms, types, and kinds. This unification is inspired by prior work on pure type systems (Cardelli 1986; Barendregt 1991; Roorda 2000).

As an example, the term 3 has type the type `Int`. Since types are integrated into the term language, `Int` is also a term, and hence has a type, namely the type "`*`", which is the "type of types". Thus, in Sage, types are simply terms of type `*`. The type `*` is also a term, and itself has type `*`.[1]

[1] Although `*` : `*` makes for inconsistent logics (Girard 1972), it does not detract from the soundness or usefulness of a type system for a programming language (Cardelli 1986).

In addition to providing a syntactically elegant language, this unification elevates types to be first-class values, which adds substantial expressive power. That is, since types are simply terms of type `*`, they can be passed to and returned from functions, just like terms of other types. [2] Thus, Sage's single construct for lambda-abstraction can express all of the following:

- normal functions, which map values to values, such as `factorial` : `Int` $\rightarrow$ `Int`.
- type operators, which map types to types, such as `ListOf` : `*` $\rightarrow$ `*` (given a type such as `Int`, this function returns the type of lists of `Int`s);
- polymorphic functions, such as the polymorphic identity function that maps a type X to a value of type $(X \rightarrow X)$;
- dependent type constructors, which are functions from values to types, such as `Range` : `Int` $\rightarrow$ `Int` $\rightarrow$ `*` (given two integers, this function returns the type of integers within that range).

Sage also supports arbitrary computations over terms, and hence over types. For example, the type of `printf` is naturally expressed as a computation that parses the first argument (the format string) to compute the expected number and type of the remaining arguments. Where possible, this computation is performed at compile time.

General Refinement Types. To express precise function pre- and post-conditions and other correctness assertions, Sage also provides *refinement types*. For example, the refinement type $\{x\!:\!\texttt{Int} \mid x > 0\}$ describes positive integers. Sage extends prior work on decidable refinement types (Xi and Pfenning 1999; Xi 2000; Freeman and Pfenning 1991; Mandelbaum et al. 2003; Ou et al. 2004) to support *general* refinement predicates — any boolean expression can be used as a refinement predicate. Thus, Sage re-uses the term language to express both types and refinement predicates.

The Type `Dynamic`. In addition to allowing programmers to document precise program invariants as types, Sage also supports dynamically-typed programming, where these invariants are omitted. Values of the special type `Dynamic` (Siek and Taha 2006; Henglein 1994; Abadi et al. 1989; Thatte 1990) are implicitly converted to and from other types as necessary.

Thus, Sage programs can use a broad range of specification paradigms, ranging from dynamically-typed Scheme-like programming, to decidable ML-like types, to precise refinement specifications. In addition, types can be incrementally added to a dynamically-typed prototype; each intermediate partially-typed program will still type-check.

[2] Sage therefore inhabits the far corner of the lambda cube (Barendregt 1991).

1.1 Hybrid Type Checking

The flexibility of dynamic typing and the generality of both first-class types and general refinement types comes at a cost: none are statically decidable.

To circumvent this limitation, SAGE replaces traditional static type checking with hybrid type checking, which enforces correctness properties and detects defects statically, whenever possible. However, hybrid type checking is willing to resort to dynamic type casts for particularly complicated situations. The overall result is that most or all types are enforced at compile time, but some complicated types may be enforced instead at run time.

We briefly illustrate the key idea of hybrid type checking by considering the function application

$$(\texttt{factorial}\ t)$$

where the function `factorial` has type $\texttt{Pos} \to \texttt{Pos}$ and $\texttt{Pos} = \{x : \texttt{Int} \mid x > 0\}$ is the type of positive integers. The behavior of the SAGE type checker depends on the type T of the argument t:

- If T can be proven to be a subtype of `Pos`, then this application is accepted as well-typed.
- Conversely, if T is a different type such as `String` that is clearly not a subtype of `Pos`, then a type error is reported.

In a conventional, decidable type system, one of these two cases always holds. Due to the expressiveness of the SAGE type system, however, we may encounter the following situations where this subtype judgment can neither be verified nor refuted:

- With first-class types, T may be a type expression that requires substantial compile-time evaluation (which SAGE supports), but this evaluation may not terminate.
- If T is a refinement type $\{x : \texttt{Int} \mid p\}$, then subtyping reduces to proving that $(p \Rightarrow x > 0)$. SAGE uses an underlying theorem prover to decide such implications where possible, but the problem is undecidable in general.
- Finally, if T is the type `Dynamic`, then SAGE cannot statically verify that the argument t is compatible with the function's domain type `Pos`.

SAGE's hybrid type checking algorithm circumvents all of these difficulties in a uniform manner. If SAGE cannot statically verify (or refute) that the argument t produces only values of the domain type `Pos`, then it inserts the type cast $\langle Pos \rangle$ on t, yielding the (well-typed) term:

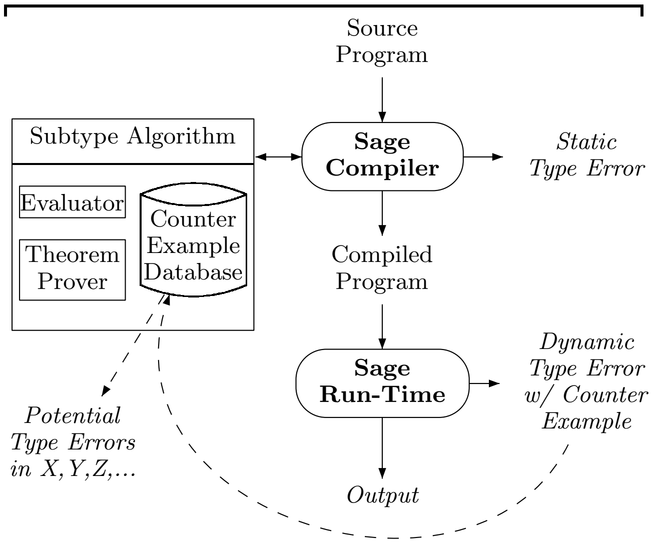
$$(\texttt{factorial}\ (\langle\texttt{Pos}\rangle\ t))$$

At run time, this term evaluates t, checks that the resulting value x satisfies the predicate $x > 0$, and only then passes that value to `factorial`. Thus, SAGE guarantees that the precise `Pos` precondition on `factorial` is always enforced, either statically or dynamically.

Note that this technique works regardless of whether the argument type T (or the domain type `Pos`) is a complex type computation, a complex refinement type, or the type `Dynamic`. Thus, hybrid type checking uniformly circumvents all of the decidability difficulties in SAGE's expressive type system.

Of course, in a traditional type system a function `factorial` of type $\texttt{Int} \to \texttt{Int}$ could internally check that its argument is positive, but this approach has several limitations: it does not document the true `factorial` interface as a type; it does not statically detect errors like `(factorial -1)`; and it may perform redundant checking in many cases.

Figure 1: Sage Architecture



Flanagan (2006) previously studied hybrid type checking in the idealized setting of the simply-typed lambda-calculus with refinements only on the base types `Int` and `Bool`. We extend hybrid type checking to the more technically challenging domain of a rich language that includes all of the features described above, and we also provide an implementation and experimental validation of this approach.

1.2 Architecture

This overall architecture of our SAGE compiler and run-time system is shown in Figure 1. It includes a subtype algorithm that integrates a compile-time evaluator (for type computations), a theorem prover (for refinement types), and also a *counter-example database*. If a compiler-inserted cast from type S to type T fails at run time, then S is not a subtype of T, and SAGE stores the fact that in this database. The type checker uses this database to subsequently reject any program that relies on S being a subtype of T. Thus, dynamic type errors can actually improve the ability of the SAGE compiler to detect type errors statically. Moreover, when a compiler-inserted cast fails, SAGE will report a list of previously-compiled programs that contain the same cast. Thus, the counter-example database functions somewhat like a regression test suite, in that it can detect errors in previously-compiled programs. Over time, the database may become a valuable repository of common but invalid subtype relationships. For performance, we also cache proven subtype relationships in the database.

1.3 Contributions

The primary contributions of this paper are as follows:

- We present SAGE, a lightweight language with a rich type system that integrates three powerful concepts: first-class types, general refinement types, and the type `Dynamic`.
- We prove the soundness of this type system.
- We present a hybrid type checking algorithm that:
  - generates only well-typed programs;
  - enforces all types, either statically or dynamically; and

  - integrates compile-time evaluation, theorem proving, and a counter-example database.

- We describe a prototype implementation of the language.
- We show that on a number of example programs and data structures, SAGE can verify the vast majority of types statically — the number of compiler-inserted casts is very small or zero in all cases.

Although our initial feasibility study with SAGE is promising, many issues remain for future work. Our implementation performs bidirectional type checking (Pierce and Turner 1998), allowing many types to be inferred locally, but does not yet perform full type inference (a much more technically challenging problem). We also plan to evaluate SAGE on larger benchmarks, to measure SAGE's ability to reject erroneous programs at compile time, and to evaluate the benefits of the counterexample database in this regard.

The presentation of our results proceeds as follows. The following section illustrates the SAGE language through a series of examples. Sections 3 and 4 define the syntax, semantics, and type system of SAGE. Section 5 presents a hybrid type checking algorithm for the language. Sections 6 and 7 describe our implementation and experimental results. Sections 8 and 9 discuss related work and future plans.

2. Motivating Examples

We introduce SAGE through several examples illustrating key features of the language, including refinement types, dependent function types, datatypes, and recursive types.

The SAGE source language extends the core language presented in Section 3 with a number of additional constructs that are desugared by the parser before type checking. In particular, the datatype construct is desugared into a collection of function definitions, via Church-style encodings.

2.1 Binary Search Trees

We begin with the commonly-studied example of binary search trees: see Figure 2. The variable `Range` has type `Int` → `Int` → `*`. Given two integers `lo` and `hi`, the application `(Range lo hi)` returns a refinement type describing integers in the range [`lo`, `hi`).

A binary search tree `(BST lo hi)` is an ordered tree containing integers in the range [`lo`, `hi`). A tree may either be `Empty`, or a `Node` containing an integer `v` ∈ [`lo`, `hi`) and two subtrees containing integers in the ranges [`lo`, `v`) and [`v`, `hi`), respectively. Thus, the type of binary search trees explicates the requirement that these trees must be ordered.

The function `search` takes as arguments two integers `lo` and `hi`, a binary search tree of type (`BST lo hi`), and an integer `x` in the range [`lo`, `hi`). Note that SAGE supports dependent function types, and so the type of the third argument to `search` can depend on the values of the first and second arguments. The function `search` then checks if `x` is in the tree. The function `insert` takes similar arguments and extends the given tree with the integer `x`.

The SAGE compiler uses an automatic theorem prover to statically verify that the specified ordering invariants on binary search trees are satisfied by these two functions — no run-time checking is required.

Precise types enable SAGE to detect various common programming errors. For example, suppose we inadvertently used the wrong conditional test:

```
24:        if x <= v
```

Figure 2: Binary Search Trees

```
 1: let Range (lo:Int) (hi:Int) : * =
 2:   {x:Int | lo <= x && x < hi };
 3:
 4: datatype BST (lo:Int) (hi:Int) =
 5:   Empty
 6: | Node of (v:Range lo hi)*(BST lo v)*(BST v hi);
 7:
 8: let rec search (lo:Int) (hi:Int) (t:BST lo hi)
 9:                (x:Range lo hi) : Bool =
10:   case t of
11:     Empty -> false
12:   | Node v l r ->
13:      if x = v then true
14:      else if x < v
15:           then search lo v l x
16:           else search v hi r x;
17:
18: let rec insert (lo:Int) (hi:Int) (t:BST lo hi)
19:                (x:Range lo hi) : (BST lo hi) =
20:   case t of
21:     Empty ->
22:      Node lo hi x (Empty lo x) (Empty x hi)
23:   | Node v l r ->
24:      if x < v
25:      then Node lo hi v (insert lo v l x) r
26:      else Node lo hi v l (insert v hi r x);
```

For this incorrect program, the SAGE compiler will report that the first recursive call to `insert` is ill-typed:

```
line 25: x does not have type (Range lo v)
```

Similarly, if an argument to `Node` is incorrect, *e.g.*:

```
26:      else Node lo hi v r (insert v hi r x);
```

the SAGE compiler will report the type error:

```
line 26: r does not have type (BST lo v)
```

A traditional type system that does not support precise specifications would not detect these errors.

Using this `BST` implementation, constructing trees with specific constraints is straightforward (and verifiable). For example, the following code constructs a tree containing only positive numbers:

```
let PosBST : * = BST 1 MAXINT;
let nil : PosBST = Empty 1 MAXINT;
let add (t:PosBST) (x:Range 1 MAXINT) : PosBST =
        insert 1 MAXINT t x;
let t : PosBST = add (add (add nil 1) 3) 5;
```

This precisely-typed BST implementation can interoperate cleanly with dynamically-typed client code, while still preserving the ordering invariant on BSTs:

```
let t1 : Dynamic = add nil 1;
let t2 : Dynamic = add t1 3;
```

2.2 Regular Expressions

We now consider more complicated types. Figure 3 declares the `Regexp` datatype and the function `match`, which determines if a string matches a regular expression. The `Regexp` datatype includes constructors to match any single letter (`Alpha`) or any single letter or digit (`AlphaNum`), as well as usual the Kleene closure, concatenation, and choice opera-

Figure 3: Regular Expressions and Names

```
datatype Regexp =
  Alpha
| AlphaNum
| Kleene of Regexp
| Concat of Regexp * Regexp
| Or of Regexp * Regexp
| Empty;

let match (r:Regexp) (s:String) : Bool = ...

let Name = {s:String | match (Kleene AlphaNum) s};
```

tors. As an example, the regular expression "`[a-zA-Z0-9]*`" is represented as `(Kleene AlphaNum)`.

The code then uses `match` to define the type `Name`, which refines the type `String` to allow only alphanumeric strings. We use the type `Name` to enforce an important, security-related interface specification for the following function `authenticate`. This function performs authentication by querying a SQL database (where '`^`' denotes string concatenation):

```
let authenticate (user:Name) (pass:Name) : Bool =
   let query : String =
     ("SELECT count(*) FROM client
       WHERE name=" ^ user ^ " and pwd=" ^ pass)
   in executeSQLquery(query) > 0;
```

This code is prone to security attacks if given specially-crafted non-alphanumeric strings. For example, calling

```
authenticate  "admin --"  ""
```

breaks the authentication mechanism because "`--`" starts a comment in SQL and consequently "comments out" the password part of the query. To prohibit this vulnerability, the type:

```
authenticate : Name → Name → Bool
```

specifies that `authenticate` should be applied only to alphanumeric strings.

Next, consider the following user-interface code:

```
let username : String = readString() in
let password : String = readString() in
authenticate username password;
```

This code is ill-typed, since it passes arbitrary user input of type `String` to `authenticate`. Proving that this code is ill-typed, however, is quite difficult, since it depends on complex reasoning showing that the user-defined function `match` is not a tautology, and hence that not all `String`s are `Name`s.

Unsurprisingly, Sage cannot statically verify or refute this code. Instead, it inserts the following casts at the call site to enforce `authenticate`'s specification dynamically:

```
authenticate (⟨Name⟩ username) (⟨Name⟩ password);
```

At run time, these casts check that `username` and `password` satisfy the predicate `match (Kleene AlphaNum)`. If the username "`admin --`" is ever entered, the cast `(⟨Name⟩ username)` will fail and halt program execution.

2.3 Counter-Example Database

Somewhat surprisingly, a dynamic cast failure actually strengthens Sage's ability to detect type errors statically. In particular, the string "`admin --`" is a witness proving that not all `String`s are `Name`s, *i.e.*, $E \not\vdash$ `String` $<:$ `Name` (where E is the typing environment for the call to `authenticate` and includes the definitions of `Regexp`, `match`, and `Name`). Rather than discarding this information, and potentially observing the same error on later runs or in different programs, such refuted subtype relationships are stored in a database. If the above code is later re-compiled, the Sage compiler will discover upon consulting this database that `String` is not a subtype of `Name`, and it will statically reject the call to `authenticate` as ill-typed.

Additionally, the database stores a list of other programs previously compiled under the assumption that `String` may be a subtype of `Name`, and Sage will also report that these programs are ill-typed. It remains to be seen how to best incorporate this unusual functionality into the software engineering process – as one example, these error reports could be inserted into a bug database for inspection at a later date.

2.4 Printf

As a final example, we examine the `printf` function. The number and type of the expected arguments to `printf` depends in subtle ways on the format string (the first argument). We can define a function

```
Args : String -> *
```

that computes the `printf` argument types for a given format string. For example, `(Args "%d%d")` evaluates to the type `Int` → `Int` → `Unit`. Using this function, we can assign to `printf` the precise type:

```
printf : (format:String -> (Args format))
```

The term `(printf "%d%d")` then has type `(Args "%d%d")`, which is evaluated at compile time to `Int` → `Int` → `Unit`. Thus, Sage is sufficiently expressive to need no special support for accommodating `printf` and catching errors in many `printf` clients statically.

Other languages, such as OCaml (Leroy et al. 2004), provide this functionality, but only for constant format strings. Some applications, however, need non-constant format strings, particularly for internationalization. Consider the following example:

```
let repeat (s:String) (n:Int) : String =
  if (n = 0) then "" else (s ^ (repeat s (n-1)));

// checked statically:
printf (repeat "%d" 2) 1 2;
```

The Sage compiler infers that `(printf (repeat "%d" 2))` has type `(Args (repeat "%d" 2))`, which evaluates (at compile time) to `Int` → `Int` → `Unit`, and hence this call is well-typed. Conversely, the compiler would statically reject the following ill-typed call:

```
// compile-time error:
printf (repeat "%d" 2) 1 false;
```

For efficiency, and to avoid non-termination, the compiler performs only a user-configurable bounded number of evaluation steps before resorting to dynamic checking. Given a small bound, the following call requires a run-time check:

```
// run-time error:
printf (repeat "%d" 20) 1 2 ... 19 false;
```

As expected, the inserted dynamic cast catches the error.

Interestingly, the type of `printf` defines an interface between two parties: the `printf` implementation and the `printf` clients. As we have seen, the client side of this interface is enforced (primarily) statically, whereas the current Sage prototype needs to enforce the *implementation* side of this interface dynamically, via the following implicitly-inserted type cast:

```
let printf (format:String) : (Args format) =
  ⟨(Args format)⟩ (...printf implementation...)
```

These static checks (on the client side) and dynamic checks (on the implementation side) co-operate to enforce both sides of the `printf` interface and to ensure type soundness. We revisit this example in more detail in Section 5 to illustrate SAGE's hybrid type checking algorithm.

3. The Core Language

3.1 Syntax

In SAGE, source programs are desugared into the small core language described in Figure 4. Although SAGE merges the syntactic categories of terms and types, we use the following naming convention to distinguish the intended use of meta-variables: s, t range over terms; x, y, z range over variables; and u, v range over values, and the corresponding capitalized variables (S, T, etc) range over types, type variables, and type values, respectively.

The core language includes variables, constants, functions, function applications, and let expressions. It also includes dependent function types, for which we use the syntax $x : S \to T$ (in preference over the equivalent notation $\Pi x : S.T$). Here, S specifies the function's domain, and the formal parameter x can occur free in the range type T. We use the shorthand $S \to T$ when x does not occur free in T.

SAGE includes the type constants `Unit`, `Bool`, `Int`, `Dynamic`, and `*`, which all have type `*`. More precise types can be introduced via the polymorphic function `Refine`. This function takes a type X and a predicate over X, and returns the *refinement type* containing all values of type X that satisfy that predicate. We use the shorthand $\{x{:}T \mid t\}$ to abbreviate `Refine` T $(\lambda x{:}T.\ t)$. Thus, $\{x{:}\texttt{Int} \mid x > 0\}$ denotes the type of positive numbers.

SAGE uses refinements to assign precise types to constants. For example, an integer n has the singleton type $\{m : \texttt{Int} \mid m = n\}$ denoting the set $\{n\}$. Similarly, the type of the operation $+$ specifies that its result is the sum of its arguments:[3]

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

The behavior of the primitive function `if` is also precisely described via polymorphic and refinement types. In particular, the "then" parameter to `if` is a thunk of type $(\{d{:}\texttt{Unit} \mid p\} \to X)$. That thunk can be invoked only if the domain $\{d : \texttt{Unit} \mid p\}$ is inhabited, *i.e.*, only if the test expression p evaluates to `true`. Thus the type of `if` precisely specifies its behavior.

The function `fix` supports recursive definitions of both functions and types; it takes a type X and a function over X, and conceptually returns a fixed point of that function. For example, the type of integer lists is defined as:

$$\texttt{fix}\ \texttt{*}\ (\lambda L{:}\texttt{*}.\ \texttt{Sum Unit}\ (\texttt{Pair Int}\ L))$$

which (roughly) returns a type L satisfying the equation:

$$L \ = \ \texttt{Sum Unit}\ (\texttt{Pair Int}\ L)$$

[3] The apparent circularity where the type of $+$ is defined in terms of $+$ itself does not cause any difficulties in our technical development, since the meaning of refinement types is defined below in terms of the operational semantics.

Figure 4: Syntax, Constants, and Shorthands

Term Syntax:

s, t, S, T	$::=$	x	variable
		c	constant
		$\texttt{let}\ x = t : S\ \texttt{in}\ t$	binding
		$\lambda x{:}S.\ t$	abstraction
		$t\ t$	application
		$x{:}S \to T$	function type

Constants:

`*`	:	`*`
`Unit`	:	`*`
`Bool`	:	`*`
`Int`	:	`*`
`Dynamic`	:	`*`
`Refine`	:	$X{:}\texttt{*} \to (X \to \texttt{Bool}) \to \texttt{*}$
`unit`	:	`Unit`
`true`	:	$\{b{:}\texttt{Bool} \mid b\}$
`false`	:	$\{b{:}\texttt{Bool} \mid \texttt{not}\ b\}$
`not`	:	$b{:}\texttt{Bool} \to \{b'{:}\texttt{Bool} \mid b' = \texttt{not}\ b\}$
n	:	$\{m{:}\texttt{Int} \mid m = n\}$
$+$	:	$n{:}\texttt{Int} \to m{:}\texttt{Int} \to \{z{:}\texttt{Int} \mid z = n + m\}$
$=$	:	$x{:}\texttt{Dynamic} \to y{:}\texttt{Dynamic} \to \{b{:}\texttt{Bool} \mid b = (x = y)\}$
`if`	:	$X{:}\texttt{*} \to p{:}\texttt{Bool} \to (\{d{:}\texttt{Unit} \mid p\} \to X) \to (\{d{:}\texttt{Unit} \mid \texttt{not}\ p\} \to X) \to X$
`fix`	:	$X{:}\texttt{*} \to (X \to X) \to X$
`cast`	:	$X{:}\texttt{*} \to \texttt{Dynamic} \to X$

Shorthands:

$$\begin{array}{rcll}
S \to T & = & x{:}S \to T & x \notin FV(T) \\
\langle T \rangle & = & \texttt{cast}\ T & \\
\{x{:}T \mid t\} & = & \texttt{Refine}\ T\ (\lambda x{:}T.\ t) & \\
\texttt{if}_T\ t_1\ \texttt{then}\ t_2\ \texttt{else}\ t_3 & = & & \\
\multicolumn{4}{r}{\texttt{if}\ T\ t_1\ (\lambda x{:}\{d{:}\texttt{Unit} \mid t\}.\ t_2)\ (\lambda x{:}\{d{:}\texttt{Unit} \mid \texttt{not}\ t\}.\ t_3)}
\end{array}$$

Here, `Sum` and `Pair` are the type constructors for Church-encoded sums and pairs, respectively:

$$\begin{array}{l}
\texttt{Pair} = \lambda X{:}\texttt{*}.\ \lambda Y{:}\texttt{*}.\ (Z{:}\texttt{*} \to (X \to Y \to Z) \to Z) \\
\texttt{Sum} = \lambda X{:}\texttt{*}.\ \lambda Y{:}\texttt{*}.\ (Z{:}\texttt{*} \to (X \to Z) \to (Y \to Z) \to Z)
\end{array}$$

The function `cast` performs run-time type casts. It takes a type X and a value of type `Dynamic` (the supertype of all types) and attempts to cast that value to type X. We use the shorthand $\langle T \rangle\ t$ to abbreviate `cast` T t. Thus, for example, the following expression casts the integer y to the refinement type of positive numbers, and fails if y is not positive.

$$\langle \{x{:}\texttt{Int} \mid x > 0\} \rangle\ y$$

3.2 Operational Semantics

We formalize the execution behavior of SAGE programs with the small-step operational semantics shown in Figure 5. Evaluation is performed inside evaluation contexts $\mathcal{E}$. Applications, let expressions, and the basic integer and boolean operations behave as expected. Rule [E-EQ] uses syntactic

Figure 5: Evaluation Rules

Evaluation $\boxed{s \longrightarrow t}$

$$\begin{array}{rcll}
\mathcal{E}[s] & \longrightarrow & \mathcal{E}[t] \quad \text{if } s \longrightarrow t & [\text{E-Compat}] \\
\\
(\lambda x{:}S.\ t)\ v & \longrightarrow & t[x := v] & [\text{E-App}] \\
\texttt{let } x = v : S \texttt{ in } t & \longrightarrow & t[x := v] & [\text{E-Let}] \\
\texttt{not true} & \longrightarrow & \texttt{false} & [\text{E-Not1}] \\
\texttt{not false} & \longrightarrow & \texttt{true} & [\text{E-Not2}] \\
\texttt{if}_T\ \texttt{true}\ v_1\ v_2 & \longrightarrow & v_1\ \texttt{unit} & [\text{E-If1}] \\
\texttt{if}_T\ \texttt{false}\ v_1\ v_2 & \longrightarrow & v_2\ \texttt{unit} & [\text{E-If2}] \\
+\ n_1\ n_2 & \longrightarrow & n \qquad n = (n_1 + n_2) & [\text{E-Add}] \\
=\ v_1\ v_2 & \longrightarrow & c \qquad c = (v_1 \equiv v_2) & [\text{E-Eq}] \\
\\
\langle\texttt{Bool}\rangle\ \texttt{true} & \longrightarrow & \texttt{true} & [\text{E-Cast-Bool1}] \\
\langle\texttt{Bool}\rangle\ \texttt{false} & \longrightarrow & \texttt{false} & [\text{E-Cast-Bool2}] \\
\langle\texttt{Unit}\rangle\ \texttt{unit} & \longrightarrow & \texttt{unit} & [\text{E-Cast-Unit}] \\
\langle\texttt{Int}\rangle\ n & \longrightarrow & n & [\text{E-Cast-Int}] \\
\langle\texttt{Dynamic}\rangle\ v & \longrightarrow & v & [\text{E-Cast-Dyn}] \\
\\
\langle\texttt{*}\rangle\ v & \longrightarrow & v & [\text{E-Cast-Type}] \\
 & & \multicolumn{2}{l}{\text{if } v \in \{\ \texttt{Int}, \texttt{Bool}, \texttt{Unit}, \texttt{Dynamic}, \texttt{*},} \\
 & & \multicolumn{2}{l}{\quad x{:}S \to T,\ \texttt{fix * } f,\ \texttt{Refine } T\ f\ \}} \\
\\
\langle x{:}S \to T\rangle\ v & \longrightarrow & & [\text{E-Cast-Fn}] \\
\multicolumn{3}{l}{\quad \lambda x{:}S.\ \langle T\rangle\ (v\ (\langle D\rangle\ x))} \\
\multicolumn{3}{l}{\qquad\qquad \text{where } D = \mathit{domain}(v)} \\
\\
\langle\texttt{Refine } T\ f\rangle\ v & \longrightarrow & \langle T\rangle\ v & [\text{E-Cast-Refine}] \\
 & & \text{if } f\ (\langle T\rangle\ v) \longrightarrow^* \texttt{true} & \\
\\
\mathcal{S}[\texttt{fix } U\ v] & \longrightarrow & \mathcal{S}[v\ (\texttt{fix } U\ v)] & [\text{E-Fix}]
\end{array}$$

$$\begin{array}{lcll}
\mathcal{E} & ::= & \bullet \mid \mathcal{E}\ t \mid v\ \mathcal{E} & \textit{Evaluation Contexts} \\
\mathcal{S} & ::= & \bullet\ v \mid \langle\bullet\rangle\ v & \textit{Strict Contexts}
\end{array}$$

$$\begin{array}{lr}
u, v, U, V ::= & \textit{Values} \\
\quad x & \text{variable} \\
\quad \lambda x{:}S.\ t & \text{abstraction} \\
\quad x{:}S \to T & \text{function type} \\
\quad c & \text{constant} \\
\quad c\ v_1\ \ldots\ v_n & \text{constant, } 0 < n < \mathit{arity}(c) \\
\quad \texttt{Refine } U\ v & \text{refinement} \\
\quad \texttt{fix } U\ v & \text{recursive type}
\end{array}$$

Figure 6: Type Rules

Environments $\boxed{E}$

$$\begin{array}{lcll}
E & ::= & \emptyset & \text{empty environment} \\
 & & E, x : T & \text{environment extension} \\
 & & E, x = v : T & \text{environment term extension}
\end{array}$$

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

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

$$\frac{(x : T) \in E \quad \text{or} \quad (x = v : T) \in E}{E \vdash x\ :\ \{y{:}T \mid\ y = x\}} \qquad [\text{T-Var}]$$

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

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

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

$$\frac{E \vdash v\ :\ S \quad E, (x = v : S) \vdash t\ :\ T}{E \vdash \texttt{let } x = v : S \texttt{ in } t\ :\ T[x := v]} \qquad [\text{T-Let}]$$

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

equality ($\equiv$) to test equivalence of all values, including function values.[4]

The most interesting reduction rules are those for casts $\langle T\rangle\ v$, which define a dynamic meaning for each type T. Casts to one of the base types `Bool`, `Unit`, or `Int` succeed if the value v is of the appropriate type. Casts to type `*` succeed only for values of type `*`. Casts to type `Dynamic` always succeed.

For casts to function types, we first introduce the function *domain*, which extracts the domain of a function value and is undefined on non-function values. Function values include λ-abstractions, partially-applied constants, and fixed point operations of function type.

$$\begin{array}{rcl}
\mathit{domain} : \mathit{Value} & \to & \mathit{Term} \\
\mathit{domain}(\lambda x{:}T.\ t) & = & T \\
\mathit{domain}(c\ v_1 \ldots v_{i-1}) & = & \text{type of } i^{th} \text{ argument to } c \\
\mathit{domain}(\texttt{fix } (x{:}T \to T')\ v) & = & T \\
\mathit{domain}(\texttt{fix } (\texttt{Refine } U\ f)\ v) & = & \mathit{domain}(\texttt{fix } U\ v)
\end{array}$$

The rule [E-Cast-Fn] casts a function v with domain type $D = \mathit{domain}(v)$ to type $x : S \to T$ by creating a new function:

$$\lambda x{:}S.\ \langle T\rangle\ (v\ (\langle D\rangle\ x))$$

This new function has the desired type $x{:}S \to T$; it takes a value x of type S, casts it to the domain type D of v, applies the given function v, and casts the result to the desired result type T. Thus, domain and range types on function casts are enforced lazily, in a manner reminiscent of higher-order contracts (Findler and Felleisen 2002) and related techniques from denotational semantics (Scott 1976).

To cast a value v to a refinement type `Refine` $T\ f$, the rule [E-Cast-Refine] first casts v to type T and then checks if the predicate f holds on this value. If it does, the cast succeeds and returns $\langle T\rangle\ v$.

The operation `fix` defines recursive functions and types, which are considered values, and hence `fix` $U\ v$ is also a value. This construct is unrolled one step to $(v\ (\texttt{fix } U\ v))$ by the rule [E-fix] whenever it appears in a *strict context* $\mathcal{S}$, *i.e.*, in a function position or in a cast.

4. The Sage Type System

Although type checking for Sage is undecidable, we can nonetheless formalize the notion of well-formed programs

[4] A semantic notion of equality could provide additional flexibility, but would be undecidable. In practice, syntactic equality has been sufficient.

Figure 7: Subtype Rules

Subtype rules $\boxed{E \vdash S <: T}$

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

$$\frac{}{E \vdash T <: \texttt{Dynamic}} \quad \text{[S-Dyn]}$$

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

$$\frac{E, F[x := v] \vdash S[x := v] <: T[x := v]}{E, x = v : U, F \vdash S <: T} \quad \text{[S-Var]}$$

$$\frac{s \longrightarrow s' \qquad E \vdash \mathcal{C}[s'] <: T}{E \vdash \mathcal{C}[s] <: T} \quad \text{[S-Eval-L]}$$

$$\frac{t \longrightarrow t' \qquad E \vdash S <: \mathcal{C}[t']}{E \vdash S <: \mathcal{C}[t]} \quad \text{[S-Eval-R]}$$

$$\frac{E \vdash S <: T}{E \vdash (\texttt{Refine}\ S\ f) <: T} \quad \text{[S-Ref-L]}$$

$$\frac{E \vdash S <: T \qquad E, x : S \models f\ x}{E \vdash S <: (\texttt{Refine}\ T\ f)} \quad \text{[S-Ref-R]}$$

via a type system. An *environment* binds variables to types and, in some cases, to values: see Figure 6. We apply implicit α-renaming to ensure that substitutions are capture-avoiding and that each variable is bound at most once in an environment.

Typing. As usual, the judgment $E \vdash t : T$ assigns type T to term t in environment E. The rules defining this judgment are mostly straightforward.

The rule [T-Const] (in Figure 6) uses the function *ty* to retrieve the type of each constant c, as defined in Figure 4. The rule [T-Var] for a variable x extracts the type T of x from the environment, and assigns to x the singleton refinement type $\{y\!:\!T \mid y = x\}$. For a function $\lambda x\!:\!S.\ t$, the rule [T-Fun] infers the type T of t and returns the dependent function type $x : S \to T$, where x may occur free in T. The term $x\!:\!S \to T$ is assigned type `*` by rule [T-Arrow], provided that both S and T have type `*`. The rule [T-App] for an application $(t_1\ t_2)$ first checks that t_1 has a function type $(x : S \to T)$ and that t_2 has type S; the application then has type T with x replaced by t_2.

The type rule [T-Let] for $\texttt{let}\ x = v : S\ \texttt{in}\ t$ first checks that v has type S, and then type checks t in the environment $E, (x = v : S)$, which contains both the type and the value of x. These value bindings are used in the subtype judgment, as described below. Subtyping is allowed at any point in a typing derivation via the rule [T-Sub].

Subtyping. The complexities and decidability issues in Sage mostly involve the subtyping judgment $E \vdash S <: T$ defined in Figure 7. The rules [S-Refl] and [S-Dyn] allow every type to be a subtype both of itself and of the type `Dynamic`. The rule [S-Fun] performs the usual contravariant/covariant checks for function subtyping.

The rules [S-Eval-L] and [S-Eval-R] support computations over types, and close the subtype relation under evaluation in arbitrary contexts $\mathcal{C}$:

$$\begin{array}{rcl} \mathcal{C} & ::= & \bullet \mid \mathcal{C}\ t \mid t\ \mathcal{C} \mid \lambda x\!:\!\mathcal{C}.\ t \mid \lambda x\!:\!T.\ \mathcal{C} \\ & & \mid x\!:\!\mathcal{C} \to T \mid x\!:\!T \to \mathcal{C} \\ & & \mid \texttt{let}\ x = \mathcal{C} : S\ \texttt{in}\ t \mid \texttt{let}\ x = t : \mathcal{C}\ \texttt{in}\ t \\ & & \mid \texttt{let}\ x = t : S\ \texttt{in}\ \mathcal{C} \end{array}$$

The rule [S-Var] facilitates this evaluation by replacing a variable with the value to which it was bound via [T-Let]. Variables with only type bindings $(x : T)$ are not substituted, so evaluation may get stuck on these variables.[5]

The final two subtype rules [S-Ref-L] and [S-Ref-R] handle refinement types on the left and right sides of the subtype relation, respectively. The rule [S-Ref-L] states that any refinement of S is a subtype of T provided that S itself is a subtype of T. The rule [S-Ref-R] states that S is a subtype of $(\texttt{Refine}\ T\ f)$ provided that S is a subtype of T and that the predicate f holds or is *valid* on all values of type S – the notion of validity is defined below.

In Sage, recursive types are introduced by the `fix` operator, whose semantics is defined via unrolling: see [E-Fix]. Hence, Sage supports a form of *equi-recursive* types (Crary et al. 1999). However, because it dramatically simplifies the metatheory, we limit our subtyping relation to the *least* fixed point of the subtype rules (all finite derivations).

Theorem Proving. The theorem proving judgment $E \models t$ states that a a boolean term t is *valid* in an environment E. We specify the interface between the type system and the theorem prover via the following axioms (akin to those used by (Ou et al. 2004)), which are sufficient to prove soundness of the type system.[6]

1. Faithfulness: If $t \longrightarrow^* \texttt{true}$ then $E \models t$. If $t \longrightarrow^* \texttt{false}$ then $E \not\models t$.
2. Hypothesis: If $(x : \{y\!:\!S \mid t\}) \in E$ then $E \models t[y := x]$.
3. Weakening: If $E, G \models t$ then $E, F, G \models t$.
4. Substitution: If $E, (x : S), F \models t$ and $E \vdash s : S$ then $E, F[x := s] \models t[x := s]$.
5. Exact Substitution: $E, (x = v : S), F \models t$ if and only if $E, F[x := v] \models t[x := v]$.
6. Preservation: If $s \longrightarrow^* t$, then $E \models \mathcal{C}[s]$ if and only if $E \models \mathcal{C}[t]$.
7. Narrowing: If $E, (x : T), F \models t$ and $E \vdash S <: T$ then $E, (x : S), F \models t$.

A consequence of the Faithfulness axiom is that the validity judgment is undecidable. In addition, the subtype judgment may require an unbounded amount of compile-time evaluation. These decidability limitations motivate the development of the hybrid type checking techniques of the following section.

Soundness. The Sage type system guarantees *progress* (*i.e.*, that well-typed programs can only get stuck due to

[5] More general techniques to permit continued evaluation in such cases might yield a larger subtype relation, but are not necessary for soundness.

[6] An alternative to these axioms is to define the validity judgment $E \models t$ directly: *i.e.*, it holds if, for all *closing substitutions* σ (from variables in E to terms consistent with their types), the term $\sigma(t)$ evaluates to true. This approach creates a non-monotonic cycle between validity and typing judgments, however, and so the consistency of the resulting system is non-obvious and remains an open question. For these reasons, we chose to axiomatize theorem proving instead.

Figure 8: Compilation Rules

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

$$\frac{(x:T) \in E \quad \text{or} \quad (x = t : T) \in E}{E \vdash x \hookrightarrow x : \{y{:}T \mid y = x\}} \quad \text{[C-Var]}$$

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

$$\frac{E \vdash S \hookrightarrow S' \downarrow * \qquad E, x : S' \vdash t \hookrightarrow t' : T}{E \vdash (\lambda x{:}S.\ t) \hookrightarrow (\lambda x{:}S'.\ t') : (x{:}S' \to T)} \quad \text{[C-Fun]}$$

$$\frac{E \vdash S \hookrightarrow S' \downarrow * \qquad E, x : S' \vdash T \hookrightarrow T' \downarrow *}{E \vdash (x{:}S \to T) \hookrightarrow (x{:}S' \to T') : *} \quad \text{[C-Arrow]}$$

$$\frac{\begin{array}{c} E \vdash t_1 \hookrightarrow t_1' : T \qquad \mathit{unrefine}(T) = x{:}S_1 \to S_2 \\ E \vdash t_2 \hookrightarrow t_2' \downarrow S_1 \end{array}}{E \vdash t_1\ t_2 \hookrightarrow t_1'\ t_2' : S_2[x := t_2']} \quad \text{[C-App1]}$$

$$\frac{\begin{array}{c} E \vdash t_1 \hookrightarrow t_1' \downarrow (\texttt{Dynamic} \to \texttt{Dynamic}) \\ E \vdash t_2 \hookrightarrow t_2' \downarrow \texttt{Dynamic} \end{array}}{E \vdash t_1\ t_2 \hookrightarrow t_1'\ t_2' : \texttt{Dynamic}} \quad \text{[C-App2]}$$

$$\frac{\begin{array}{c} E \vdash S \hookrightarrow S' \downarrow * \qquad E \vdash v \hookrightarrow v' \downarrow S' \\ E, (x = v' : S') \vdash t \hookrightarrow t' : T \qquad T' = T[x := v'] \end{array}}{\begin{array}{c} E \vdash \texttt{let}\ x = v : S\ \texttt{in}\ t \\ \hookrightarrow \texttt{let}\ x = v' : S'\ \texttt{in}\ t' : T' \end{array}} \quad \text{[C-Let]}$$

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

$$\frac{E \vdash t \hookrightarrow t' : S \qquad E \vdash^{\surd}_{alg} S <: T}{E \vdash t \hookrightarrow t' \downarrow T} \quad \text{[CC-Ok]}$$

$$\frac{E \vdash t \hookrightarrow t' : S \qquad E \vdash^{?}_{alg} S <: T}{E \vdash t \hookrightarrow (\langle T \rangle\ t') \downarrow T} \quad \text{[CC-Chk]}$$

Subtyping algorithm (see Fig. 9) $\boxed{E \vdash^{a}_{alg} S <: T}$

failed casts) and *preservation* (*i.e.*, that evaluation of a term preserves its type). The proofs appear in the appendix.

5. Hybrid Type Checking

In Sage, subtyping, and hence type checking, is undecidable. Sage circumvents this limitation using hybrid type checking, which is based on a subtype algorithm that conservatively approximates the undecidable subtyping relation. For a given subtype query $E \vdash S <: T$, this subtype algorithm $E \vdash^{a}_{alg} S <: T$ returns a result $a \in \{\surd, \times, ?\}$ indicating whether:

($\surd$) the algorithm succeeds in proving the query;
($\times$) the algorithm refutes the query; or
(?) the algorithm cannot decide this particular query.

Compilation. Using this conservative subtype algorithm, we define a hybrid type checking or *compilation* algorithm

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

that type checks the source program s in environment E and inserts dynamic casts to compensate for limitations in the subtype algorithm, yielding the well-typed term t of type T. The compilation and checking judgment

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

is similar, except that it takes as an input the desired type T and ensures that t has that type.

The rules defining these judgments are shown in Figure 8. Many of these rules are similar to the corresponding type rules, *e.g.*, [C-Var] and [C-Const]. The rule [C-Fun] for $\lambda x{:}S.\ t$ compiles S to some type S' of type $*$ and compiles t to a term t' of type T, and returns the compiled function $\lambda x{:}S'.\ t'$ of type $x{:}S' \to T$. The rule [C-Arrow] for a function type compiles the two component types, which must have type $*$. The rule [C-Let] compiles the term $\texttt{let}\ x = v : S\ \texttt{in}\ t$ by recursively compiling v, S and t in appropriate environments.

The rules for a function application $t_1\ t_2$ are more interesting. The rule [C-App1] starts by compiling the function t_1 to some term t_1' of some type T. This type T may not be a syntactic function type; instead it may be a refinement of a function type, or it may require evaluation to yield a function type. The following partial function $\mathit{unrefine} : \mathit{Term} \to \mathit{Term}$ extracts the underlying syntactic function type in these cases:

$$\begin{array}{rcll} \mathit{unrefine}(x{:}S \to T) & = & x{:}S \to T & \\ \mathit{unrefine}(\texttt{Refine}\ T\ f) & = & \mathit{unrefine}(T) & \\ \mathit{unrefine}(T) & = & \mathit{unrefine}(T') & \text{if } T \longrightarrow T' \end{array}$$

Since the evaluation performed by *unrefine* may not terminate, the rule [C-App2] provides a backup strategy for applications that performs fewer static checks and more dynamic checks. This rule checks that the function t_1 has only the most general function type `Dynamic` $\to$ `Dynamic`, and correspondingly coerces the argument t_2 to type `Dynamic`, resulting in an application with type `Dynamic`.

Compilation and Checking. The rules defining the compilation and checking judgment $E \vdash s \hookrightarrow t \downarrow T$ illustrate the key ideas of hybrid type checking. These rules compile the given term and check that the compiled term has the expected type T via the subtyping query

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

If this query succeeds ($a = \surd$), then [CC-OK] returns the compiled term. If the query is undecided ($a = ?$), then [CC-Chk] encloses the compiled term in the cast $\langle T \rangle$ to preserve dynamic type safety. If the query fails ($a = \times$), then no rule applies and the program is rejected as ill-typed.

Compilation of `Dynamic`. The type system (via [S-Dyn]) and the operational semantics (via [E-Cast-Dyn]) both treat `Dynamic` as a maximal or top type. As a consequence, the program

$$P \overset{\text{def}}{=} \lambda\, add1{:}(\texttt{Int} \to \texttt{Int}).\ \lambda x{:}\texttt{Dynamic}.\ (add1\ x)$$

is technically ill-typed, as `Dynamic` is not a subtype of `Int`.

To permit convenient interoperation between statically and dynamically typed code, however, we would like Sage to accept this program and to implicitly downcast `Dynamic` to the domain type `Int`. To achieve this desired behavior, we simply have the subtype algorithm return the unknown result $E \vdash^{?}_{alg}$ `Dynamic` $<:$ `Int`, which causes the compilation rules [C-App1] and [CC-Chk] to accept this program and insert the desired downcast, yielding the well-typed program:[7]

$$\lambda\, add1 : (\texttt{Int} \to \texttt{Int}).\ \lambda x : \texttt{Dynamic}.\ (add1\ (\langle \texttt{Int} \rangle\ x))$$

We generalize this requirement for the subtype algorithm as the following lemma.

LEMMA 1 (SUBTYPE ALGORITHM).

1. *If $E \vdash^{\surd}_{alg} S <: T$ then $E \vdash S <: T$.*
2. *If $E \vdash^{\times}_{alg} S <: T$ then $\forall E', S', T'$ that are obtained from E, S, T by replacing the type* `Dynamic` *by any type, we have that $E' \not\vdash S' <: T'$.*

Clearly, a naïve subtype algorithm could always return the result "?" and thus trivially satisfy these requirements. The following section describes a more precise subtype algorithm that enables SAGE to verify more properties and to detect more errors at compile time.

We note that the type $\texttt{Dynamic} \to \texttt{Dynamic}$ mentioned earlier is the most general function type with respect to this subtype algorithm (although not the subtype relation). In particular, for any function type U (which may require unbounded evaluation to yield a syntactic function type $x : S \to T$), we have that $E \vdash^{a}_{alg} U <: (\texttt{Dynamic} \to \texttt{Dynamic})$ for some $a \in \{\surd, ?\}$.

Example. To illustrate how SAGE verifies specifications statically when possible, but dynamically when necessary, consider the `printf` client:

$$t \overset{\text{def}}{=} \texttt{printf "\%d" 4}$$

For this term, the rule [C-APP1] will first compile the subexpression `(printf "%d")` via the following compilation judgment (based on the type of `printf` from Section 2.4):

$$\ldots \vdash \texttt{(printf "\%d")} \hookrightarrow \texttt{(printf "\%d")} : \texttt{(Args "\%d")}$$

The rule [C-APP1] then calls the function *unrefine* to evaluate `(Args "%d")` to the normal form $\texttt{Int} \to \texttt{Unit}$. Since 4 has type `Int`, the term t is therefore accepted as is; no casts are needed.

Alternatively, if the computation for `(Args "%d")` does not terminate within a preset time limit, the compiler uses the rule [C-APP2] to compile t into the code:

$$(\langle \texttt{Dynamic} \to \texttt{Dynamic} \rangle\ \texttt{(printf "\%d")})\ \ 4$$

At run time, `(printf "%d")` evaluates to some function $(\lambda x : \texttt{Int}.\ \cdots)$ that expects an `Int`, yielding the application:

$$(\langle \texttt{Dynamic} \to \texttt{Dynamic} \rangle\ (\lambda x : \texttt{Int}.\ \cdots))\ \ 4$$

The rule [E-CAST-FN] then reduces this term to:

$$(\lambda x : \texttt{Dynamic}.\ \langle \texttt{Dynamic} \rangle\ ((\lambda x : \texttt{Int}.\ \cdots)\ (\langle \texttt{Int} \rangle\ x)))\ \ 4$$

where the nested cast $\langle \texttt{Int} \rangle\ x$ dynamically ensures that the next argument to `printf` must be an integer.

Correctness. Even though the SAGE type system is undecidable, the compilation algorithm is guaranteed to generate only well-typed programs. The proof appears in the appendix. The compilation algorithm may accept some complex but ill-typed programs, but it will insert sufficient type casts to enforce all types and to ensure that the compiled program is well-typed. Hence, compiled programs only go wrong on type casts that were either inserted by the programmer or by the compiler for ill-typed source programs.

[7] Interestingly, the compilation algorithm makes explicit this check that a Scheme implementation would perform implicitly inside the primitive $add1 : \texttt{Dynamic} \to \texttt{Dynamic}$.

Typing Source Programs. The SAGE type system is applicable to both the source and target programs of a compilation judgment $E \vdash s \hookrightarrow t : T$. For the target program, the type system guarantees preservation and progress. For the source program, however, this type system is a little incomplete in its handling of `Dynamic`. For example, the type system does not accept the program P on page 8, because it does not support implicit conversion from `Dynamic` to `Int`.

We could overcome this limitation by defining a source program to be well-typed if it compiles using the most precise subtype algorithm satisfying Lemma 1. From this definition, we can derive a separate collection of typing rules for source programs that permit implicit conversions between `Dynamic` and other types, and which adapt the ideas of Siek and Taha (2006) to our more complicated language. Space limitations preclude the presentation of this additional source-language type system here, and, in any case, our existing type system functions adequately for our purposes.

6. Implementation

Our prototype SAGE implementation consists of roughly 6,000 lines of OCaml code. It extends the core language of Section 3 with a number of additional constructs that are desugared by the parser, before type checking. It also performs bidirectional type checking (Pierce and Turner 1998), allowing many types to be inferred locally.

Evaluation. The SAGE run time implements the semantics from Section 3, plus a counter-example database of failed casts. Specifically, suppose the compiler inserts the cast $(\langle T \rangle\ t)$ because it cannot prove or refute some subtype test $E \vdash S <: T$. If that cast fails on a value v, the run time inserts an entry into the database asserting that $E \not\vdash S <: T$, and records v as a witness of this fact.

To correctly assign blame for cast failures, every cast carries a label identifying the corresponding term in the source program, and these labels are propagated as necessary during evaluation. In particular, the two new casts inserted by [E-CAST-FN] carry the same label as the original function cast. These labels are then sufficient to provide precise blame assignment — in particular, the more complex blame assignment techniques for higher-order contracts (Findler and Felleisen 2002) are unnecessary for higher-order type casts (Gronski and Flanagan 2007).

Subtype Algorithm. The key difficulty in implementing hybrid type checking for SAGE is in providing an adequately precise subtype algorithm. The SAGE subtype algorithm applies the rules of Figure 9, in the order in which they are presented, and it supports equi-recursive types by computing the greatest fixed point of these rules. These rules rely on the 3-valued conjunction operator $\otimes$:

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

The rules [AS-REFL], [AS-FUN], [AS-DYN-R], [AS-REF-L], and [AS-VAR] are straightforward adaptations of corresponding earlier rules. The rule [AS-DYN-L] ensures that converting from `Dynamic` to any other type requires an explicit coercion.

The rules [AS-EVAL-L] and [AS-EVAL-R] perform evaluation of types within $\mathcal{D}$-contexts. To avoid non-terminating computations (*e.g.*, due to the infinite unrolling of recursive

Figure 9: Subtyping Algorithm

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

$$\frac{E \vdash^\times_{db} S <: T}{E \vdash^\times_{alg} S <: T} \quad \text{[AS-DB]}$$

$$\frac{}{E \vdash^{\surd}_{alg} T <: T} \quad \text{[AS-REFL]}$$

$$\frac{E \vdash^a_{alg} T_1 <: S_1 \quad E, x : T_1 \vdash^b_{alg} S_2 <: T_2 \quad c = a \otimes b}{E \vdash^c_{alg} (x{:}S_1 \to S_2) <: (x{:}T_1 \to T_2)} \quad \text{[AS-FUN]}$$

$$\frac{}{E \vdash^?_{alg} \texttt{Dynamic} <: T} \quad \text{[AS-DYN-L]}$$

$$\frac{}{E \vdash^{\surd}_{alg} S <: \texttt{Dynamic}} \quad \text{[AS-DYN-R]}$$

$$\frac{E \vdash^a_{alg} S <: T \quad a \in \{\surd, ?\}}{E \vdash^a_{alg} (\texttt{Refine}\ S\ f) <: T} \quad \text{[AS-REF-L]}$$

$$\frac{E \vdash^a_{alg} S <: T \quad E, x : S \models^b_{alg} f\ x \quad c = a \otimes b}{E \vdash^c_{alg} S <: (\texttt{Refine}\ T\ f)} \quad \text{[AS-REF-R]}$$

$$\frac{E, F[x := v] \vdash^a_{alg} S[x := v] <: T[x := v]}{E, x = v : U, F \vdash^a_{alg} S <: T} \quad \text{[AS-VAR]}$$

$$\frac{s \longrightarrow s' \quad E \vdash^a_{alg} \mathcal{D}[s'] <: T}{E \vdash^a_{alg} \mathcal{D}[s] <: T} \quad \text{[AS-EVAL-L]}$$

$$\frac{t \longrightarrow t' \quad E \vdash^a_{alg} S <: D_2[t']}{E \vdash^a_{alg} S <: D_2[t]} \quad \text{[AS-EVAL-R]}$$

$\mathcal{D} ::= \bullet \mid N\ \mathcal{D}$ where N is a normal form

Algorithmic theorem proving $\boxed{E \models^a_{alg} t}$

separate algorithm

Counter-example database $\boxed{E \vdash^\times_{db} S <: T}$

database of previously failed casts

types), the algorithm bounds the number of applications of these two rules.

If no rule is applicable, then the algorithm returns "×" provided that both types are values; if either type is not a value, the algorithm returns the conservative result "?".

Counter-Example Database. The subtype rule [AS-DB] attempts to refute $E \vdash S <: T$ by querying the counter-example database. It uses the judgment $E \vdash^\times_{db} S <: T$, which essentially looks for an exact match for the triple $\langle E, S, T\rangle$ in the database. To maximize the likelihood of a database hit, the triple is first translated into an equivalent but canonical form by (1) removing from E any bindings for variables that are not directly or transitively referenced by S or T, and (2) by replacing all variable names by deBruijn indices. Database entries are also translated into canonical form before insertion.

Theorem Proving Algorithm. The rule [AS-REF-R] for checking whether S is a subtype of a specific refinement type relies on the theorem proving algorithm $E \models^a_{alg} t$, which conservatively approximates the validity judgment $E \models t$. Like the subtype algorithm, the theorem proving algorithm returns a 3-valued result $a \in \{\surd, ?, \times\}$, and includes special treatment for the type `Dynamic`:

REQUIREMENT 2 (THEOREM PROVING ALGORITHM).

1. *If $E \models^{\surd}_{alg} t$ then $E \models t$.*
2. *If $E \models^\times_{alg} t$ then $\forall E', t'$ obtained from E and t by replacing the type* `Dynamic` *by any type, we have that $E' \not\models t'$.*

Our current theorem proving algorithm translates the query $E \models^a_{alg} t$ into input for the Simplify theorem prover (Detlefs et al. 2005). For example, the query

$$x : \{x{:}\texttt{Int} \mid x \geq 0\} \models^a_{alg} x + x \geq 0$$

is translated into the Simplify query:

```
(IMPLIES (>= x 0) (>= (+ x x) 0))
```

for which Simplify returns `Valid`, and so $a = \surd$.

For more complex queries that cannot be expressed in Simplify's input language (involving recursive definitions, etc), our algorithm simply returns "?".

One interesting issue arises with queries such as:

$$x : \texttt{Int} \models^a_{alg} x * x \geq 0$$

Simplify fails to prove this query, but since Simplify is incomplete for arbitrary multiplication, we return "?" instead of "×". The theorem proving algorithm returns "×" for a query only if Simplify is complete on that query and still fails to find a proof. We currently assume that Simplify is complete for linear integer arithmetic. Simplify has very effective heuristics for integer arithmetic, but does not fully satisfy this specification; we plan to replace it with an alternative prover that is complete for this domain.

7. Experimental Results

We evaluated the SAGE language, type system, and implementation on a number of example programs. The program `arith.sage` defines and uses a number of mathematical functions, such as `min`, `abs`, and `mod`, where refinement types provide precise specifications. The programs `bst.sage` and `heap.sage` implement and use binary search trees and heaps, and the program `polylist.sage` defines and manipulates polymorphic lists. The types of these data structures ensure that every operation preserves key invariants. The program `stlc.sage` implements a type checker and evaluator for the simply-typed lambda calculus (STLC), where SAGE types specify that evaluating an STLC-term preserves its STLC-type. We also include the sorting algorithm `mergesort.sage`, as well as the `regexp.sage` and `printf.sage` examples discussed earlier.

Figure 10 characterizes the performance of the subtype algorithm on these benchmarks. We consider two configurations of this algorithm, both with and without the theorem prover. For each configuration, the figure shows the number of subtyping judgments proved (denoted by $\surd$), or left undecided (denoted by ?) — the benchmarks are all well-typed, so no subtype queries are refuted (denoted by $\times$). Note that the theorem prover enables SAGE to decide many more subtype queries. In particular, many benchmarks include complex refinement types that use integer arithmetic to specify ordering and structure invariants; theorem proving is particularly helpful in verifying these benchmarks.

Figure 10: Subtyping Algorithm Statistics

Benchmark	Lines of code	Without Prover ✓	?	×	With Prover ✓	?	×
`arith.sage`	45	132	13	0	145	0	0
`bst.sage`	62	344	28	0	372	0	0
`heap.sage`	69	322	34	0	356	0	0
`mergesort.sage`	80	437	31	0	468	0	0
`polylist.sage`	397	2338	5	0	2343	0	0
`printf.sage`	228	321	1	0	321	1	0
`regexp.sage`	113	391	2	0	391	2	0
`stlc.sage`	227	677	11	0	677	11	0
Total	1221	4962	125	0	5073	14	0

Our subtyping algorithm performs quite well and verifies a large majority of subtype tests performed by the compiler. Only a small number of undecided queries result in casts. For example, in `regexp.sage`, SAGE cannot statically verify subtyping relations involving regular expressions (they are checked dynamically) but it statically verifies all other subtype judgments. Some complicated tests in `stlc.sage` must also be checked dynamically. As described earlier, `printf.sage` includes a single cast that enforces the specification of `printf`.

Despite the use of a theorem prover, type checking times for these benchmarks are quite manageable. On a 3GHz Pentium 4 Xeon processor running Linux 2.6.14, type checking required fewer than 10 seconds for each of the benchmarks, except for `polylist.sage` which took approximately 18 seconds. We also measured the number of evaluation steps required during each subtype test. We found that 83% of the subtype tests required no evaluation, 91% required five or fewer steps, and only a handful of the the tests in our benchmarks required more than 50 evaluation steps.

Many opportunities remain for further improvement, both in the SAGE implementation itself, and in its application to more substantial programs. Nevertheless, this preliminary study suggests that, even though SAGE supports an expressive and undecidable type system, the techniques of this paper are sufficiently precise to support practical programming with these rich types.

8. Related Work

The enforcement of complex program specifications, or *contracts*, is the subject of a large body of prior work (Parnas 1972; Meyer 1988; Luckham 1990; Findler and Felleisen 2002; Blume and McAllester 2004; Findler and Blume 2006). Since these contracts are typically not expressible in classical type systems, they have previously been relegated to dynamic checking, as in, for example, Eiffel (Meyer 1988), whose expressive contract language is strictly separated from its type system. Hybrid type checking extends contracts with the ability to check many properties at compile time. Meunier *et al* have also investigated statically verifying contracts via set-based analysis (Meunier et al. 2006).

Recent work on advanced type systems has influenced our choice of how to express program invariants. In particular, Freeman and Pfenning (1991) extended ML with another form of refinement types, and Xi and Pfenning (1999) have explored applications of dependent types in Dependent ML. 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. Our system of dependent types extends theirs with arbitrary executable refinement predicates, and the hybrid type checking infrastructure copes with the resulting undecidability. In a complementary approach, Chen and Xi (2005) address decidability limitations by providing a mechanism through which the programmer can provide proofs of subtle properties in the source code.

Recently, Ou et al. (2004) developed a dependent type system that also leverages dynamic checks. In comparison to SAGE, their system is less expressive but decidable, and they leverage dynamic checks to reduce the need for precise type annotations in explicitly labeled regions of programs.

Barendregt (1991) used the unification of types and terms to allow computations over types while simplifying the underlying theory. The language Cayenne adopts this approach and copes with the resulting undecidability of type checking by allowing a maximum number of compile-time evaluation steps before reporting to the user that typing has failed (Augustsson 1998). Hybrid type checking differs in that instead of rejecting subtly well-typed programs outright, it provisionally accepts them and then performs dynamic checking where necessary.

Concurrent with this work, Siek and Taha (2006) introduced a similar notion of `Dynamic` that enables clean interoperation between statically- and dynamically-typed code. Their initial work was for the simply-typed lambda-calculus, and they have since extended these ideas to support objects (Siek and Taha 2007). Other authors have considered pragmatic combinations of both static and dynamic checking (Abadi et al. 1989; Henglein 1994; Thatte 1990). For Scheme, soft type systems (Fagan 1990; Wright and Cartwright 1994; Aiken et al. 1994) prevent some basic type errors statically, while checking other properties at run time.

The limitations of purely-static and purely-dynamic approaches have also motivated other work on hybrid analyses. For example, CCured (Necula et al. 2002) is a sophisticated hybrid analysis for preventing the ubiquitous array bounds violations in C programs. Although the static analysis was initially used only to optimize the run-time analysis, it has recently been extended with the ability to detect errors at compile time (Condit et al. 2007).

The static checking tool ESC/Java (Flanagan et al. 2002) supports expressive JML specifications (Leavens and Cheon 2005). However, ESC/Java's error messages may be caused either by incorrect programs or by limitations in its own analysis, and thus it may give false alarms on correct (but perhaps complicated) programs. The Spec# programming system extends C# with expressive specifications (Barnett et al. 2005), which are enforced dynamically, and can be also checked statically via a separate tool. The system is somewhat less tightly integrated than in SAGE, and so static verification does not automatically remove corresponding dynamic checks.

9. Conclusions and Future Work

This paper explores an unusual approach to the design of a typed programming language. In contrast to traditional decidable type systems, SAGE's type system is a synthesis of three highly expressive yet undecidable concepts: first-class types, general refinement types, and the type `Dynamic`. The SAGE compiler integrates several techniques (theorem proving, compile-time evaluation, and a counter-example database) that perform static type checking on a "best effort" basis, but relies on dynamic type casts to enforce soundness in particularly complicated situations.

Overall, this design works quite well. The resulting language is small yet surprisingly powerful. It supports very precise type specifications, but does not demand them; omitted type annotations default to `Dynamic`. Computation and function abstraction work equivalently and cleanly on both types and terms. A notion of reflection is provided for free, since types are first-class values. All types, including complex function and refinement types, can be enforced via run-time casts, and static and dynamic type checks co-operate closely to ensure type soundness. Experimental results show that SAGE can verify most or all correctness properties at compile time.

Many opportunities remain for future work. We plan to integrate randomized or directed (Godefroid et al. 2005) testing to refute additional validity queries, thereby detecting more errors at compile time. The benefits of the counter-example database can be amplified by maintaining a single (perhaps distributed, peer-to-peer) repository for all users of SAGE. We also plan to investigate type inference for SAGE, perhaps leveraging the flexibility of the type `Dynamic` in particularly complicated situations. Finally, we hope to adapt ideas from SAGE to a more mainstream language, perhaps by extending the Glasgow Haskell compiler (Jones et al. 1993).

Acknowledgments

We thank Robby Findler and Bo Adler for useful feedback on this work. This work was supported by the National Science Foundation under Grants CCR-0341179 and CCR-0341387. Additional support was provided through a Sloan Fellowship.

References

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

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

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

H. Barendregt. Introduction to generalized type systems. *Journal of Functional Programming*, 1(2):125–154, 1991.

M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. volume 3362 of *Lecture Notes in Computer Science*, pages 49–69, 2005.

M. Blume and D. A. McAllester. A sound (and complete) model of contracts. In *International Conference on Functional Programming*, pages 189–200, 2004.

L. Cardelli. A polymorphic lambda calculus with type:type. Technical Report 10, DEC Systems Research Center, Palo Alto, California, 1986.

C. Chen and H. Xi. Combining programming with theorem proving. In *International Conference on Functional Programming*, pages 66–77, 2005.

J. Condit, M. Harren, Z. Anderson, D. Gay, and G. C. Necula. Dependent types for low-level programming. In *European Symposium on Programming*, 2007.

K. Crary, R. Harper, and S. Puri. What is a recursive module? In *Conference on Programming Language Design and Implementation*, pages 50–63, 1999.

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

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

R. B. Findler and M. Blume. Contracts as pairs of projections. In *Symposium on Logic Programming*, pages 226–241, 2006.

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

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

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

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

J.-Y. Girard. *Interprétation fonctionelle et élimination des coupures dans l'arithmétique d'ordre supérieure.* PhD thesis, University of Paris, 1972.

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

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

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

S. L. P. Jones, C. V. Hall, K. Hammond, W. Partain, and P. Wadler. The Glasgow Haskell compiler: a technical overview. In *Joint Framework for Information Technology Technical Conference*, pages 249–257, 1993.

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

X. Leroy, D. Doligez, J. Garrigue, D. Rémy, and J. Vouillon. The Objective Caml system, release 3.08. `http://caml.inria.fr/pub/docs/manual-ocaml/`, 2004.

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

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

P. Meunier, R. B. Findler, and M. Felleisen. Modular set-based analysis from contracts. In *Symposium on Principles of Programming Languages*, pages 218–231, 2006.

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

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

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.

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

B. C. Pierce and D. N. Turner. Local type inference. In *Symposium on Principles of Programming Languages*, pages 252–265, 1998.

J.-W. Roorda. Pure type systems for functional programming. Master's thesis, Utrecht University, 2000.

D. Scott. Data types as lattices. *SIAM Journal on Computing*, 5(3):522–587, 1976.

J. G. Siek and W. Taha. Gradual typing for functional languages. In *Scheme and Functional Programming Workshop*, September 2006.

J. G. Siek and W. Taha. Gradual typing for objects. In *European Conference on Object-Oriented Programming*, 2007. (to appear).

S. Thatte. Quasi-static typing. In *Symposium on Principles of Programming Languages*, pages 367–381, 1990.

A. Wright and R. Cartwright. A practical soft type system for Scheme. In *Conference on Lisp and Functional Programming*, pages 250–262, 1994.

H. Xi. Imperative programming with dependent types. In *IEEE Symposium on Logic in Computer Science*, pages 375–387, 2000.

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

Appendix

We present the formal development of Sage and the proofs stated earlier in the paper in this Appendix. In Appendix A, we state the key correctness theorems for Sage. In Appendix B, we prove a number of useful properties regarding the Sage subtype relation. We prove the standard progress and preservation theorems for the Sage type system in Appendix C and D, respectively. Finally, we prove the soundness of Sage's hybrid type checking strategy in Appendix E.

A. Theorem Statements

We prove correctness of Sage's type system and semantics with respect to an augmented typing relation $E \vdash_r t : T$ that extends the relation $E \vdash t : T$ with the following rule, allowing any closed term to be given any correct refinement type:

<u>T</u>ype rules $\boxed{E \vdash_r t : T}$

$$\frac{\emptyset \vdash_r s : T \qquad \emptyset \vdash_r \{x{:}T \mid t\} : * \qquad \emptyset \models t[x := s]}{\emptyset \vdash_r s : \{x{:}T \mid t\}} \quad \text{[T-Refine]}$$

This rule (while not necessarily useful in the original type system) is essential for proving that types are preserved under evaluation. Moreover, for technical reasons, including it in the original type system would prevent us from guaranteeing that the compilation process compiles all well-typed programs.

We also formalize well-formed environments as follows. The only unusual aspect of our environments is that we sometimes have values bound to the names in in the environment, in which case those values must be well-typed.

<u>W</u>ell-formed <u>e</u>nvironment $\boxed{\vdash E}$

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

$$\frac{\vdash E \qquad E \vdash T : *}{\vdash E, x : T} \quad \text{[We-Ext1]}$$

$$\frac{\vdash E \qquad E \vdash T : * \qquad E \vdash v : T}{\vdash E, x = v : T} \quad \text{[We-Ext2]}$$

In the following statement of soundness, a *failed cast* is a term of the form $\langle T \rangle\ v$ that is a normal form.

Theorem 3 (Progress). *Suppose $\emptyset \vdash_r t : T$. If t is a normal form then it is either a value or contains a failed cast.*

Theorem 4 (Preservation). *If $\emptyset \vdash_r s : T$ and $s \longrightarrow t$ then $\emptyset \vdash_r t : T$.*

Theorem 5 (Soundness of Hybrid Type Checking).

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

These three theorems are proved in Appendices C–E below.

B. Preliminary Lemmas

This section states and proves several properties of type system used in the rest of the development. Throughout this appendix, we make use of our axiomatization of theorem proving, described on page 7. For convenience, we repeat those axioms here.

Property 6 (Theorem Proving Axioms). *The theorem proving judgment $E \models t$ conforms to the following axioms:*

1. Faithfulness: *If $t \longrightarrow^* \texttt{true}$ then $E \models t$. If $t \longrightarrow^* \texttt{false}$ then $E \not\models t$.*
2. Hypothesis: *If $(x : \{y{:}S \mid t\}) \in E$ then $E \models t[y := x]$.*
3. Weakening: *If $E, G \models t$ then $E, F, G \models t$.*
4. Substitution: *If $E, (x : S), F \models t$ and $E \vdash s : S$ then $E, F[x := s] \models t[x := s]$.*
5. Exact Substitution: *$E, (x = v : S), F \models t$ if and only if $E, F[x := v] \models t[x := v]$.*

6. Preservation: *If $s \longrightarrow^* t$, then $E \models \mathcal{C}[s]$ if and only if $E \models \mathcal{C}[t]$.*
7. Narrowing: *If $E, (x : T), F \models t$ and $E \vdash S <: T$ then $E, (x : S), F \models t$.*

B.1 Substitution

This first core lemma must be proved simultaneously for a variety of judgments.

Lemma 7 (Substitution). *Suppose E and F are environments; t, s, S, and T are terms (types). If $E, x : S, F \vdash_r t : T$ and $E \vdash_r s : S$, and $\vdash E, x : S, F$ then the following statements hold (where $\theta = [x := s]$, and if the environment contains $x = v : S$ then we require that $s = v$)*

1. If $\vdash E, x : S, F$ then $\vdash E, \theta F$
2. If $E, x : S, F \vdash T <: U$ then $E, \theta F \vdash \theta T <: \theta U$.
3. If $E, x : S, F \vdash_r t : T$ then $E, \theta F \vdash_r \theta t : \theta T$.

Proof: The proof proceeds by induction, using a rather complex induction scheme. The "outermost" induction is on the length of E. Since the use of this inductive process is limited to a very small part of the proof, we do not duplicate the parts of the proof that are independent of the length of E.

The proof then proceeds by mutual induction on the derivations in the antecedents (even though the environment grows in some subderivations, the prefix E does not, so the nested induction is well-founded)

This structural induction is lexicographic in the following sense: For parts 3 and 2 assume that part 1 holds for the input environment, even though it does not appear as a subderivation, while in the proof of part 1 we assume only that parts 2 and 3 hold for strict subderivations.

1. Now we proceed with the inductive cases:
   - [We-Empty]: Trivial
   - [We-Ext1] or [We-Ext2]: Then either $F = \emptyset$, in which case the conclusion of the lemma is the antecedents of the rule, or $F = F', y : S$, in which case the lemma holds by induction on each antecedent.
2. Assume $E, x : S, F \vdash T <: U$ and proceed by induction on the derivation:
   - [S-Refl], [S-Dyn]: Trivial.
   - [S-Fun], [S-Ref-L], [S-Ref-R], [S-Eval]: These cases follow immediately from the inductive hypothesis.
   - [S-Var]: There are several cases to consider, depending on whether the definition used is in E or F or is x itself.
     - The definition is in E : We then have:
       $$E = E_1, y = v : R, E_2$$
       $$E_1, \pi E_2, x : \pi S, \pi F \vdash \pi T <: \pi U$$
       where $\pi = [y := v]$.

       Since $\vdash E$ we know that $E_1 \vdash_r v : R$. By induction, using part (1), we then have that $\vdash E_1, \pi E_2, x : \pi S, \pi F$ We will now also use the induction on E; because $|E_1| < |E|$ the substitution lemma holds for $[y := v]$ so
       $$E_1, \pi E_2 \vdash_r \pi s : \pi S$$
       Letting $\vartheta = [x := \pi s]$ allows us to conclude by induction on the subderivation that
       $$E_1, \pi E_2, \vartheta(\pi F) \vdash \vartheta(\pi T) <: \vartheta(\pi U)$$
       Note that v can be typed without x in the environment, so $x \notin FV(v)$. Thus,
       $$\begin{aligned} \vartheta \circ \pi &= [x := \pi s] \circ [y := v] \\ &= [y := \vartheta v] \circ [x := \pi^2 s] \\ &= [y := v] \circ [x := \pi^2 s] && \text{because } x \text{ is not free in } v \\ &= \pi \circ [x := \pi s] = \pi \circ \theta && \text{because } \pi \text{ is idempotent} \end{aligned}$$
       Then, substituting according to this equality,
       $$E_1, \pi E_2, \pi(\theta F) \vdash \pi(\theta T) <: \pi(\theta U)$$
       Then from [S-Var]
       $$E_1, y = v : R, E_2, \theta F \vdash \theta T <: \theta U$$
       which is exactly
       $$E, \theta F \vdash \theta T <: \theta U$$
     - The definition is in F : In this case we have that
       $$F = F_1, y = v : R, F_2$$
       $$E, x : S, F_1, \pi F_2 \vdash \pi T <: \pi U$$

By part 1 we know $\vdash E, x : S, F_1, \pi F_2$, so letting $\varpi = [y := \theta v]$, we have by induction,

$$E, \theta F_1, \theta(\pi F_2) \vdash \theta(\pi T) <: \theta(\pi U)$$

Note that $y \notin FV(s)$ because s typed without y in the environment. Thus,

$$\begin{array}{lll} \theta \circ \pi & = & [x := s] \circ [y := v] \\ & = & [x := s] \circ [y := \theta v] \\ & = & [y := \theta v] \circ [x := \varpi s] \\ & = & \varpi \circ \theta \qquad\qquad \text{because } y \text{ not free in } s \end{array}$$

And substituting according to this,

$$E, \theta F_1, \varpi(\theta F_2) \vdash \varpi(\theta T) <: \varpi(\theta F)$$

Hence by [S-Var]

$$E, \theta F_1, y = \theta v : R, \theta F_2 \vdash \theta T <: \theta U$$

Which is exactly

$$E, \theta F \vdash \theta T <: \theta U$$

- The definition is $x = v : S$: Then we have

$$v = s$$
$$E \pi F \vdash \pi T <: \pi U$$

The second statement is exactly the desired conclusion.

3. Consider the final rule applied in $E, x : S, F \vdash_r t : T$.

- [T-Var]: $t = y$ for some variable y. There are several cases to consider:
  - $y = x$: In this case,

$$t = y = x$$

  and applying θ yields

$$\theta t = \theta y = s$$
$$T = S = \theta S = \theta T$$

  Thus,

$$E \vdash_r s : S \quad \equiv \quad E \vdash_r \theta t : \theta T$$

  and $E, \theta F \vdash_r \theta t : \theta T$ follows by weakening.
  - $y \neq x$: In this case, $\theta t = \theta y = y$. There are several cases to consider:
    - $y : T \in E$: By applying θ, we have

$$\theta T = T$$

    and it follows that

$$y : \theta T \in E, \theta F$$

    By rule [T-Var],

$$E, \theta F \vdash_r y : T$$

    which is $E, \theta F \vdash_r \theta t : \theta T$.
    - $y : T \in F$: Applying θ yields

$$y : \theta T \in E, \theta F$$

    which permits us to conclude via rule [T-Var] that

$$E, \theta F \vdash_r y : \theta T$$

    which is $E, \theta F \vdash_r \theta t : \theta T$.

- [T-Sub]: In this case,

$$E, x : S, F \vdash_r t : S$$
$$E, x : S, F \vdash S <: T$$

  By induction,

$$E, \theta F \vdash_r \theta t : \theta S$$

  and using statement 2 we can apply statement 2 to conclude that

$$E, \theta F \vdash \theta S <: \theta T$$

  Thus, by rule [T-Sub], we can derive

$$E, \theta F \vdash_r \theta t : \theta T$$

- [T-Refine]: In this case,

  $$\begin{array}{l} T = \{y{:}S \mid p\} \\ E, x : S, F \vdash_r t \;:\; S \\ E, x : S, F \vdash_r S \;:\; * \\ E, x : S, F \models p[y := t] \end{array}$$

  By induction,

  $$\begin{array}{l} E, \theta F \vdash_r \theta t \;:\; \theta S \\ E, \theta F \vdash_r \theta S \;:\; * \end{array}$$

  We may then use Property 6 (Substitution) to show that

  $$E, \theta F \models \theta(p[y := t])$$

  which can be rewritten as

  $$E, \theta F \models (\theta p)[y := \theta t]$$

  It then follows from rule [T-Refine] that

  $$E, \theta F \vdash_r \theta t \;:\; \theta T$$

- [T-Const]: Trivial.
- [T-Fun], [T-Arr], [T-Let], [T-App]: These cases follow immediately from the inductive hypothesis.

□

B.2 Properties of Subtyping

A first simple property is that subtyping respects definitions in the environment

Lemma 8 (Reflexivity of subtyping under exact substitution). *Let $\theta = [x := v]$. For any types S, T and environments E, F, if $E \vdash v \;:\; T$ then $E, x = v : T, F \vdash S <: \theta S$ and $E, x = v : T, F \vdash \theta S <: S$*

Proof: As substitutions are idempotent, $\theta S = \theta^2 S$ so by [S-Refl] $E, \theta F \vdash \theta S <: \theta^2 S$. Then by [S-Var] we have $E, y = v : T, F \vdash S <: \theta S$, and the symmetric relationship by an analogous proof. □

Next we prove Narrowing (that when a type in the environment is made "smaller" the subtyping judgment still holds)

Lemma 9 (Narrowing of Subtyping).
If $\vdash E, x : B, F$ and $E, x : B, F \vdash S <: T$ and $E \vdash A <: B$ then $E, x : A, F \vdash S <: T$.
If $(x = v : B)$ is in the environment, then we additionally require $E \vdash v \;:\; A$.

Proof: We proceed by induction on the derivation of $E, x : B, F \vdash S <: T$, considering the final rule applied.

- [S-Refl], [S-Dyn]: Trivial.
- [S-Fun]: From the hypotheses of this rule, it must be that

  $$S = y : S_1 \to S_2 \qquad T = y : T_1 \to T_2 \qquad E, x : B, F \vdash T_1 <: S_1 \qquad E, x : B, F, y : T_1 \vdash S_2 <: T_2$$

  By induction, we know that $E, x : A, F \vdash T_1 <: S_1$ and $E, x : A, F, x : T_1 \vdash S_2 <: T_2$. By [S-Fun], we can conclude that $E, x : A, F \vdash S <: T$.
- [S-Ref-L]: From the hypotheses of this rule, it must be that

  $$S = \texttt{Refine}\ W\ f \qquad E, x : B, F \vdash W <: T$$

  By induction, we know that $E, x : A, F \vdash W <: T$. Then by application of [S-Ref-L] we can conclude that $E, x : A, F \vdash S <: T$ holds.
- [S-Ref-R]: From the hypotheses of this rule, it must be that

  $$T = \texttt{Refine}\ W\ f \qquad E, x : B, F \vdash S <: W \qquad E, x : B, F, x : S \models f\ x$$

  By induction, we know that $E, x : A, F \vdash S <: W$. By the Narrowing Axiom of the theorem proving judgment $E, x : A, F, x : S \models f\ x$ holds. These two facts applied to [S-Ref-R] rule allows us to conclude that $E, x : A, F \vdash S <: T$.
- [S-Eval-L] and [S-Eval-R]: These cases follow easily from the inductive hypothesis.
- [S-Var]: From the hypotheses of this rule, it must be that

  $$(y = v : R) \in E, x : V, F$$

  so let $\theta = [y := v]$ and there are three cases to consider:

  - $(y = v : R) \in F$: this case is immediate by induction

- $y = x$: In this case, $T = B$ and

$$(x = v : B) \in E, x = v : B, F \qquad E, \theta F \vdash \theta S <: \theta V$$

Then simply apply [S-Var] to conclude $E, x = v : A, F \vdash S <: V$ (the environment is well-formed because we require $E \vdash v : A$). This is the same as $E, x = v : A, F \vdash S <: T$

- $(y = v : R) \in E$:
Since the environment is well-formed, we know that $E = E_1, y = v : R, E_2$ where $E_1 \vdash v : R$. From the antecedent of [S-Var], we have $E_1, \theta E_2, x : \theta B, \theta F \vdash \theta S <: \theta T$. By Lemma 7 (Substitution) we have that $E_1, \theta E_2 \vdash \theta A <: \theta B$
Applying induction, we narrow the environment to conclude $E_1, \theta E_2, x : \theta A, \theta F \vdash \theta S <: \theta T$ and then apply [S-Var] to finish with

$$E, x : A, F \vdash S <: T$$

□

Lemma 10 (Preservation of Subtyping Under Evaluation). *For any environment E, if $t \longrightarrow^* t'$ then*

1. *$E \vdash \mathcal{C}[t] <: \mathcal{C}[t']$ and*
2. *$E \vdash \mathcal{C}[t'] <: \mathcal{C}[t]$.*

Proof: We proceed to show (1) by induction on the derivation of $t \longrightarrow^* t'$. If zero evaluation steps are performed, then $t = t'$ and $E \vdash C[t] <: C[t']$ follows via rule [S-Refl]. The inductive case follows from rule [S-Eval-R], which shows that the necessary subtyping relationship is preserved by a single evaluation step. Case (2) is similar, but uses rule [S-Eval-L] in the inductive case. □

Subtyping is also transitive, as shown below.

Lemma 11 (Transitivity of Subtyping). *If $\vdash E$ and $E \vdash S <: T$ and $E \vdash T <: U$ then $E \vdash S <: U$.*

Proof: We proceed by induction on the derivation of $E \vdash T <: U$ and consider each possible rule used in the last step of that derivation:

- [S-Refl]: In this case, $T = U$, so we may immediately conclude that $E \vdash S <: U$.
- [S-Dyn]: In this case, we have that $U = \texttt{Dynamic}$, and $E \vdash S <: U$ follows via rule [S-Dyn].
- [S-Fun]: In this case, the following four statements must be true:

$$T = x : T_1 \to T_2 \qquad U = x : U_1 \to U_2 \qquad E \vdash U_1 <: T_1 \qquad E, x : U_1 \vdash T_2 <: U_2$$

We now proceed by induction on the derivation of $E \vdash S <: T$ and consider each possible rule used in the last step of that derivation:

  - [S-Refl]: Trivial.
  - [S-Dyn]: T must be `Dynamic`, which contradicts the statement that $T = x : T_1 \to T_2$. Therefore, this case cannot happen.
  - [S-Var]: In this case,

$$E = E_1, x = v : W, E_2 \qquad E_1, E_2[x := v] \vdash S[x := v] <: T[x := v]$$

By Lemma 7 (Substitution), we can also conclude that

$$E_1, E_2[x := v] \vdash T[x := v] <: U[x := v]$$

By induction,

$$E_1, E_2[x := v] \vdash S[x := v] <: U[x := v]$$

and by rule [S-Var], we have that

$$E \vdash S <: U$$

  - [S-Eval-L]: In this case,

$$S = \mathcal{C}[s] \qquad s \longrightarrow s' \qquad E \vdash \mathcal{C}[s'] <: T$$

By induction, $E \vdash \mathcal{C}[s'] <: U$, and we may conclude that $E \vdash S <: T$ via rule [S-Eval-L].
  - [S-Eval-R]: Similar to previous case.
  - [S-Fun]: In this case,

$$S = x : S_1 \to S_2 \qquad E \vdash T_1 <: S_1 \qquad E, x : T_1 \vdash S_2 <: T_2$$

By induction, we may conclude that $E \vdash U_1 <: S_1$ holds. Since $E \vdash U_1 <: T_1$, Lemma 9 indicates that $E, x : U_1 \vdash S_2 <: U_2$. These two statements enable us to conclude that $E \vdash x : S_1 \to S_2 <: x : U_1 \to U_2$ via rule [S-Fun].

  - [S-Ref-L]: In this case,
$$S = (\texttt{Refine}\ V\ f) \qquad E \vdash V <: T$$
and we may conclude that $E \vdash V <: U$ by induction. Rule [S-Ref-L] then indicates that $E \vdash (\texttt{Refine}\ V\ f) <: U$.
  - [S-Ref-R]: In this case, $T = \texttt{Refine}\ V\ f$, but this cannot occur because $(\texttt{Refine}\ V\ f) \neq x : T_1 \rightarrow T_2$.

  Thus, regardless of how we derive that $E \vdash S <: T$, we may conclude $E \vdash S <: U$.
- [S-Var]: In this case,
$$E = E_1, x = v : W, E_2 \qquad E_1, E_2[x := v] \vdash T[x := v] <: U[x := v]$$
By Lemma 7, we can also conclude that
$$E_1, E_2[x := v] \vdash S[x := v] <: T[x := v]$$
By induction,
$$E_1, E_2[x := v] \vdash S[x := v] <: U[x := v]$$
and by rule [S-Var], we have that
$$E \vdash S <: U$$
- [S-Eval-L]: In this case,
$$T = \mathcal{C}[t] \qquad t \longrightarrow t' \qquad E \vdash \mathcal{C}[t'] <: U$$
We now proceed by induction on the derivation of $E \vdash S <: T$, and consider each possible rule used in the last step of that derivation. All are similar to other cases presented above and below.
- [S-Eval-R]: In this case,
$$U = \mathcal{C}[t] \qquad t \longrightarrow t' \qquad E \vdash T <: \mathcal{C}[t']$$
By induction, $E \vdash S <: \mathcal{C}[t']$, and we may conclude that $E \vdash S <: U$ via rule [S-Eval-R].
- [S-Ref-L]: In this case, the following statements must be true:
$$T = \texttt{Refine}\ V\ f \qquad E \vdash V <: U$$
We now proceed by induction on the derivation of $E \vdash S <: T$ and consider each possible rule used in the last step of that derivation:
  - [S-Refl], [S-Dyn], [S-Var], [S-Eval-R], and [S-Eval-L]: Similar to above.
  - [S-Fun]: In this case, $T = x : T_1 \rightarrow T_2$, which contradicts the assumption that $T = \texttt{Refine}\ V\ f$. Therefore, this case cannot happen.
  - [S-Ref-L]: In this case,
$$S = (\texttt{Refine}\ W\ f') \qquad E \vdash W <: T$$
and we may conclude that $E \vdash W <: U$ by induction. Rule [S-Ref-L] then indicates that $E \vdash (\texttt{Refine}\ W\ f') <: U$.
  - [S-Ref-R]: In this case, since we have assumed $T = (\texttt{Refine}\ V\ f)$, we know that
$$E \vdash S <: V \qquad E, x : S \models f\ x$$
By induction, $E \vdash V <: U$, and we may conclude that $E \vdash (\texttt{Refine}\ V\ f) <: U$ via rule [S-Ref-L].

  Thus, regardless of how we derive that $E \vdash S <: T$, we may conclude $E \vdash S <: U$.
- [S-Ref-R]: In this case, the following statements must be true:
$$U = \texttt{Refine}\ V\ f \qquad E \vdash T <: V \qquad E, x : T \models f\ x$$
By the induction, we know that $E \vdash S <: V$. Lemma 9 implies that $E, x : S \models f\ x$. Thus, $E \vdash S <: \texttt{Refine}\ V\ f$ follows via rule [S-Ref-R].

Thus, transitivity holds for all possible derivations of the two hypotheses. □

Corollary 12. *Suppose $\vdash E$, then*

1. *If $E \vdash \mathcal{C}[s] <: T$ and $s \longrightarrow s'$ then $E \vdash \mathcal{C}[s'] <: T$.*
2. *If $E \vdash S <: \mathcal{C}[t]$ and $t \longrightarrow t'$ then $E \vdash S <: \mathcal{C}[t']$.*

Proof: Follows from Lemma 10 and Lemma 11. □

Subtyping is also preserved under environment weakening. This relies on our assumption the theorem proving judgment $E \models t$ is also preserved under weakening, as described on page 7.

Lemma 13 (Weakening of subtyping). *For all environments E, F, and G with pairwise disjoint domains, and types S and T, if $E, G \vdash S <: T$ then $E, F, G \vdash S <: T$.*

PROOF: By induction on the derivation $E, G \vdash S <: T$, and case analysis of the last rule used.

- [S-REF-R]:: As hypotheses to that rule,

  $T = \{x{:}T_1 \mid t\}$
  $E, G, x : S \models t$
  $E, G \vdash S <: T_1$

  By induction, we may conclude that

  $E, F, G \vdash S <: T_1$

  From Property 6 (Weakening), we have

  $E, F, G, x : S \models t$

  which then enables us to show via rule [S-REF-R] that

  $E, F, G \vdash S <: T$

- [S-REFL] and [S-DYN]:: Trivial, since the environment is irrelevant.
- [S-FUN], [S-REF-L], [S-EVAL-L], [S-EVAL-R], and [S-VAR]: These follow immediately from the inductive hypothesis.

□

The next technical lemma describes properties of the *unrefine* function, which strips outer refinements from an inner function or base type.

Finally, we show that, given a type T, removing outer refinements from T via the function *unrefine*(T) yields a supertype of T. For example, `Int` is a supertype of $\{x{:}\texttt{Int} \mid f\}$.

LEMMA 14 (UNREFINE). *For all U and E, if unrefine(U) is defined, then $E \vdash U <:$ unrefine(U).*

PROOF: We proceed by induction on the computation of *unrefine*(U):

- $U = \texttt{Refine}\ T\ f$: In this case, *unrefine*(U) = *unrefine*(T). By the induction hypothesis, $E \vdash T <:$ *unrefine*(T) holds. By rule [S-REF-L], we know that $E \vdash \texttt{Refine}\ T\ f <: T$, and by Lemma 11, it must be that $E \vdash \texttt{Refine}\ T\ f <:$ *unrefine*(T). Hence $E \vdash U <:$ *unrefine*(U).
- $\exists U'$ such that $U \rightarrow U'$: In this case, *unrefine*(U) = *unrefine*(U'). By the induction hypothesis, $E \vdash U' <:$ *unrefine*(U') holds. Using the empty context in Lemma 10 we may conclude that $E \vdash U <: U'$. Together, we may infer that $E \vdash U <:$ *unrefine*(U') via Lemma 11, and thus that $E \vdash U <:$ *unrefine*(U).
- $U = x{:}S \rightarrow T$: In this case, *unrefine*$(U) = x{:}S \rightarrow T$. By rule [S-REFL], $E \vdash x{:}S \rightarrow T <: x{:}S \rightarrow T$.

□

The next two lemmas show inversion properties of subtyping. The first shows that the standard covariant and contravariant subtyping relationships hold for the components of two related function types. The second states a similar property for refinement types.

LEMMA 15 (INVERSION OF FUNCTION SUBTYPING). *If $\vdash E$ and $E \vdash x{:}S_1 \rightarrow S_2 <: x{:}T_1 \rightarrow T_2$ then $E \vdash T_1 <: S_1$ and $E, x : T_1 \vdash S_2 <: T_2$.*

PROOF: By induction on the subtyping derivation $E \vdash x{:}S_1 \rightarrow S_2 <: x{:}T_1 \rightarrow T_2$.

- [S-DYN], [S-REF-L], and [S-REF-R]: These rules cannot conclude the desired statement.
- [S-REFL]: Then $T_1 = S_1$ and $T_2 = S_2$ so we can apply [S-REFL] to each.
- [S-FUN]: The antecedents of the rule are exactly the desired conclusions.
- [S-VAR]: Then we know $E = E_1, y = v : U, E_2$ and $E_1, \theta E_2 \vdash \theta(x{:}S_1 \rightarrow S_2) <: \theta(x{:}T_1 \rightarrow T_2)$ where $\theta = [y := v]$.

  By induction, distributing θ over the arrows, we have that $E_1, \theta E_2 \vdash \theta T_1 <: \theta S_1$ and can apply [S-VAR] to conclude $E \vdash T_1 <: S_1$. In similar fashion, $E, x : T_1 \vdash S_2 <: T_2$ follows immediately from the inductive hypothesis.
- [S-EVAL-R], and [S-EVAL-L]: We consider [S-EVAL-L]. The only evaluation rule that can apply to $x : S_1 \rightarrow S_2$ is [E-COMPAT] so suppose evaluation proceeds by evaluating $S_1 \longrightarrow S_1'$, and that $E \vdash x{:}S_1' \rightarrow S_2 <: x{:}T_1 \rightarrow T_2$. Then by induction we know that $E \vdash T_1 <: S_1'$, and applying [S-EVAL-R] we obtain $E \vdash T_1 <: S_1$. Evaluation in other contextual positions and the case for [S-EVAL-L] are analogous.

  □

LEMMA 16 (INVERSION OF REFINEMENT SUBTYPING). *If $\vdash E$ and $E \vdash S <: \{x{:}T \mid p\}$ then $E \vdash S <: T$ and $E, x : S \models p$.*

PROOF: We proceed by induction on the subtyping derivation, considering the last rule used.

- [S-Refl]: Given that $S = \{x\!:\!T \mid p\}$, we may conclude that $E, x : \{y\!:\!T \mid p\} \models p$ by Property 6 (Hypothesis). It then follows that $E \vdash S <: T$ via rules [S-Refl] and [S-Ref-L].
- [S-Dyn], [S-Fun]: These cases cannot happen, given the hypotheses.
- [S-Ref-R]: Trivial.
- [S-Ref-L]: In this case,

  $$\begin{array}{l} S = \{y\!:\!S_1 \mid q\} \\ E \vdash S_1 <: \{x\!:\!T \mid p\} \end{array}$$

  By induction, we conclude that

  $$\begin{array}{l} E, x : S_1 \models p \\ E \vdash S_1 <: T \end{array}$$

  Using rules [S-Refl] and [S-Ref-L], we may conclude that $E \vdash S <: S_1$ and by Property 6 (Narrowing), $E, x : S \models p$. We may then conclude that $E \vdash S <: T$ via rule [S-Ref-L].
- [S-Eval-L]: In this case,

  $$\begin{array}{l} S = \mathcal{C}_S[s] \\ s \longrightarrow s' \\ E \vdash \mathcal{C}_S[s'] <: \{x\!:\!T \mid p\} \end{array}$$

  By induction, we know that

  $$\begin{array}{l} E, x : \mathcal{C}_S[s'] \models p \\ E \vdash \mathcal{C}_S[s'] <: T \end{array}$$

  Lemma 10 (Preservation of Subtyping under Evaluation) indicates that $E \vdash S <: \mathcal{C}_S[s']$, and Property 6 (Narrowing) enables us to conclude that $E, x : S \models p$. Rule [S-Eval-L] then shows $E \vdash \mathcal{C}_S[s] <: T$.
- [S-Eval-R]:

  $$\begin{array}{l} \{x\!:\!T \mid p\} = \mathcal{C}_T[t] \\ t \longrightarrow t' \\ E \vdash S <: \mathcal{C}_T[t'] \end{array}$$

  In this case, the hole in $\mathcal{C}_T$ appears either in T or in p.

  - The hole appears in T:
    Then by induction,

    $$E \vdash S <: T[t']$$

    We may then conclude via rule [S-Eval-R] that

    $$E \vdash S <: T[t]$$

  - The hole appears in p:
    Then by induction,

    $$E, x : S \models p[t']$$

    Using Property 6 (Preservation), we have that

    $$E, x : S \models p[t]$$

- [S-Var] : As hypotheses to this rule,

  $$\begin{array}{l} E = E_1, y = v : U, E_2 \\ E_1, E_2[y := v] \vdash S[y := v] <: \{x\!:\!T[y := v] \mid p[y := v]\} \end{array}$$

  By induction,

  $$\begin{array}{l} E_1, E_2[y := v], x : S[y := v] \models p[y := v] \\ E_1, E_2[y := v] \vdash S[y := v] <: T[y := v] \end{array}$$

  By rule [S-Var],

  $$E \vdash S <: T$$

  which then allows us to conclude via Property 6 (Exact Substitution) that

  $$E, x : S \models p$$

□

C. Progress

We now turn our attention to showing that evaluation of well-typed terms proceeds until we are left with a value or encounter a failed cast. We first characterize all possible expression forms that are evaluated at run time, as well as the types that they may have. The canonical types (those which cannot be evaluated) are defined as follows.

Definition 17 (Canonical Types). *A term T is a* canonical type *if it is in one of the following forms:* `*`, `Unit`, `Int`, `Bool`, $x : S \rightarrow U$, `Refine` $S\ t$, *or* `Dynamic`.

All values encountered at run time can be assigned one of these canonical types, as stated in the following lemma. This lemma also defines the set of values V_T belonging to each canonical type T.

Lemma 18 (Canonical Shapes). *If $\emptyset \vdash_r v : T$, and v is a normal form, then it must be the case that $\emptyset \vdash_r T : *$ and that T is a canonical type. For each T there is also a fixed set of canonical shapes V_T, such that if $\emptyset \vdash_r v : T$ then $v \in V_T$. The sets V_T are defined as follows:*

1. $V_{*} = \{*, \texttt{Int}, \texttt{Bool}, \texttt{Unit}, x:S \rightarrow U, \texttt{Refine}\ S\ t, \texttt{fix}\ *\ v, \texttt{Dynamic}\}$.
2. $V_{\texttt{Int}} = \{n \mid n \in Z\}$.
3. $V_{\texttt{Bool}} = \{\texttt{true}, \texttt{false}\}$.
4. $V_{\texttt{Unit}} = \{\texttt{unit}\}$.
5. $V_{xS\rightarrow T} = \{\lambda x : S'.\ t$, `not`, `if`, `if` U, `if` $U\ b$, `if` $U\ b\ v$, $+$, $+\ n$, `eq`, `eq` U, `eq` $U\ v_1$, `cast`, `cast` U, `fix`, `fix` U, `Refine`, `Refine` $U\}$, *where* $b \in \{\texttt{true}, \texttt{false}\}$.
6. $V_{\texttt{Refine}\ T\ t} = V_T$ *(by definition).*
7. $V_{\texttt{Dynamic}} =$ *all values.*

Proof: We proceed by induction on the derivation of $\emptyset \vdash v : T$. Note that the inductive hypothesis is *not* strengthened to allow arbitrary environments.

- [T-Var], [T-Let]: This case cannot occur, since this rule does not assign types to values.
- [T-Var]:, This rule cannot apply because the environment is empty.
- [T-Const]: Here, v can be any constant from Figure 4:
  - If $v \in \{*, \texttt{Unit}, \texttt{Bool}, \texttt{Int}, \texttt{Dynamic}\}$, then $T = *$.
  - If $v = \texttt{unit}$ then $T = \texttt{Unit}$.
  - If $v \in \{\texttt{true}, \texttt{false}\}$ then $T = \texttt{Refine}\ \mathit{Bool}\ t$.
  - If $v \in \{n \mid n \in Z\}$ then $T = \texttt{Refine}\ \mathit{Int}\ t$.
  - Otherwise, $T = x : S \rightarrow T$.
- [T-Fun]: Here, v must be $\lambda x : S.\ t$ and T must be $x : S \rightarrow T$ (there is no induction on the body of the function, because the canonical shape is already determined by the rule)
- [T-Arrow]: Here, v must be $x : S \rightarrow T$ and T must be $*$.
- [T-App]: If $v = \texttt{Refine}\ T'\ v$ or $v = \texttt{fix}\ *\ v$ then $T = *$. Otherwise, $T = x : S \rightarrow T$.
- [T-Refine]: We have that $V_{\texttt{Refine}\ T\ t} = V_T$, so the theorem holds by the inductive hypothesis.
- [T-Sub]: We know that $\emptyset \vdash_r v : S$, so by the induction hypothesis we know that v has one of the canonical shapes of type S. We must show that if $\emptyset \vdash S <: T$, then $V_S \subseteq V_T$. We proceed by induction on the derivation of $\emptyset \vdash S <: T$:
  - [S-Refl]: Trivial, since $S = T$.
  - [S-Dyn]: V_T is the set of all values, so V_S must be included in it.
  - [S-Fun]: S and T have the same shape, so $V_S = V_T$.
  - [S-Ref-L]: Let $S = \texttt{Refine}\ S'\ s$, since S must be a refinement type. By the antecedent of [S-Ref-L], $\emptyset \vdash S' <: T$ so $V_{S'} \subseteq V_T$. By induction, $V_S = V_{\texttt{Refine}\ S'\ f} = V_{S'}$, so $V_S \subseteq V_T$.
  - [S-Ref-R]: Let $T = \texttt{Refine}\ T'\ t$, since T must be a refinement type. By the antecedent of [S-Ref-R], $\emptyset \vdash S <: T'$, so $V_S \subseteq V_{T'}$. By induction, $V_T = V_{\texttt{Refine}\ T'\ f} = V_{T'}$, and it then follows that $V_S \subseteq V_T$.
  - [S-Eval-L]: We know that $\emptyset \vdash C[s'] <: T$ by the antecedent of [S-Eval-L]. Therefore, by induction, $V_{C[s']} \subseteq V_T$. By Lemma 10 (Preservation of Subtyping) and induction, we then know that $V_{C[s]} = V_{C[s']}$. Thus, $V_{C[s]} \subseteq V_T$.
  - [S-Eval-R]: We know that $\emptyset \vdash S <: C[t']$ by the antecedent of [S-Eval-R]. Therefore, by induction, $V_S \subseteq V_{C[t']}$. By Lemma 10 (Preservation of Subtyping), and induction, we then know that $V_{C[t]} = V_{C[t']}$. Thus, $V_S \subseteq V_{C[t]}$.
  - [S-Var]: This case cannot occur, since E is empty.

□

t_1	Applicable Evaluation Rule
$\lambda x\!:\!S.\ t$	[E-App]
`not`	[E-Not1] or [E-Not2]
`if`	value
`if` U	value
`if` U v_1	value
`if` U `true` v	[E-If1]
`if` U `false` v	[E-If2]
$+$	value
$+\ n$	[E-Add]
`eq`	value
`eq` U	value
`eq` U v	[E-Eq]
`Refine`	value
`Refine` U	value
`fix`	value
`fix` U	[E-Fix]
`cast`	value
`cast` U	one of [E-Cast-...], or none if the cast fails

Table 1. Applicable evaluation rules for $t_1\ t_2$. (If $t_1\ t_2$ is a value, then no rule will apply.)

The previous two lemmas enable us to now prove that evaluation of well-typed programs proceeds until a value is reached or a failed cast is encountered. Specifically, the only normal forms are values of terms containing a failed cast.

Restatement of Theorem 3 (Progress) *Suppose $\emptyset \vdash_r t : T$. If t is a normal form then it is either a value or contains a failed cast.*

Proof: We proceed by induction on a derivation of $\emptyset \vdash_r t : T$. Assuming that $\emptyset \vdash_r t : T$, and that progress holds for all of the subderivations of t, we show that it holds for t:

- [T-Var]: In this case, $t = x$. However, this is not possible if $\emptyset \vdash_r t : T$.
- [T-Const]: In this case, $t = c$ and $t : ty(c)$, and we already have a value.
- [T-Fun]: In this case, $t = \lambda x\!:\!S.\ t$ and $T = (x\!:\!S \to T)$ and $S : *$, and we already have a value.
- [T-Arrow]: In this case, $t = x\!:\!S \to T$ and $T = *$ and $S : *$. Thus, we already have a value.
- [T-Let]: In this case, $t = \texttt{let}\ x = t_1 : T_1\ \texttt{in}\ t_2$.
  - If t_1 is not a value, then the context rule [E-Compat] allows t_1 to be evaluated. It follows from induction that progress holds for t_1.
  - If t_1 is a value, then [E-Let] applies.
- [T-Sub]: In this case, $t = t$ and the theorem holds by the inductive hypothesis.
- [T-App]: In this case,

$$\begin{array}{l} t = t_1\ t_2 \\ \emptyset \vdash_r t_1 : x\!:\!S \to T \\ \emptyset \vdash_r t_2 : S \end{array}$$

  There are three cases to consider:
  - t_1 is not a value: The context rule [E-Compat] allows t_1 to be evaluated, and we know by the inductive hypothesis that progress holds for t_1.
  - t_1 is a value but t_2 is not a value: The context rule [E-Compat] allows t_2 to be evaluated, and we know by the inductive hypothesis that progress holds for t_2.
  - t_1 and t_2 are both values: Evaluation can proceed as given in Table 1. Since application is well-typed, by rule [T-App], we know that t_1 must have some function type $x\!:\!S \to T$. Hence, t_1 must be one of the values specified in Lemma 18 (Canonical Shape - part 5). Table 1 shows that for each possible value t_1, either
    - $t_1\ t_2$ is a value,
    - $t_1\ t_2$ can be reduced by the given rule,
    - or $t_1\ t_2$ is a failed cast.
- T-Refine: In this case, $t = t$, an the theorem holds by the inductive hypothesis.

□

D. Preservation

This section shows that an expression's type is preserved under evaluation. We begin with the basic fact that typing is insensitive to the addition of bindings to the environment. First, typing is preserved under environment weakening.

LEMMA 19 (WEAKENING OF TYPING). *For all environments E, F, and G such that $\vdash E, G$ and $\vdash E, F, G$, if $E, G \vdash_r t : T$ then $E, F, G \vdash_r t : T$.*

PROOF: We proceed by induction on the structure of the derivation of $E, G \vdash_r t : T$ and case analysis of the last rule used:

- [T-CONST], [T-VAR]: Trivial.
- [T-FUN], [T-ARROW], [T-APP], [T-LET]: The conclusion follows immediately from the inductive hypothesis.
- [T-SUB]: In this case,

  $$E, G \vdash_r t : S$$
  $$E, G \vdash S <: T$$

  By induction,

  $$E, F, G \vdash_r t : S$$

  According to Lemma 13 (Weakening of Subtyping),

  $$E, F, G \vdash S <: T$$

  We may then use rule [T-SUB] to conclude

  $$E, F, G \vdash_r t : T$$

- [T-REFINE]: In this case,

  $$T = \{x : S \mid p\}$$
  $$E, G \vdash_r t : S$$
  $$E, G \vdash_r \{x : S \mid p\} : *$$
  $$E, G \models p[x := t]$$

  By induction,

  $$E, F, G \vdash_r t : S$$
  $$E, F, G \vdash_r \{x : S \mid p\} : *$$

  Property 6 (Weakening) then shows that

  $$E, F, G \models p[x := t]$$

  which allows to conclude via rule [T-REFINE] that

  $$E, F, G \vdash_r t : T$$

□

LEMMA 20. *If $E \vdash_r \lambda x : S_1.\ t : x : T_1 \rightarrow T_2$ then $E \vdash T_1 <: S_1$.*

PROOF: By induction on the derivation of $E \vdash_r \lambda x : S_1.\ t : x : T_1 \rightarrow T_2$; only [T-LAM] (where the lemma is immediate) and [T-SUB] (where the lemma follows by induction) apply. □

RESTATEMENT OF THEOREM 4 (PRESERVATION) *If $\emptyset \vdash_r s : T$ and $s \longrightarrow t$ then $\emptyset \vdash_r t : T$.*

PROOF: We proceed by induction on the structure of the derivation $E \vdash_r s : T$ and perform case analysis on the last rule of that derivation:

- [T-VAR, T-CONST, T-FUN, T-ARROW]: There is no evaluation rule for s, so the theorem is trivially true.
- [T-SUB]: In this case,

  $$\exists S.\ \emptyset \vdash S <: T \text{ and } \emptyset \vdash_r s : S$$

  By induction,

  $$\emptyset \vdash_r t : S$$

  Rule [T-SUB] then allows us to conclude

  $$\emptyset \vdash_r t : T$$

- [T-APP]: In this case,

  $$s = s_1\ s_2$$
  $$\emptyset \vdash_r s_1 : x : T_1 \rightarrow T_2$$
  $$\emptyset \vdash_r s_2 : T_1$$
  $$T = T_2[x := s_2]$$

There are several cases to consider:

- s_1 is not a value: Since evaluation is strict, as expressed by the grammar for evaluation contexts $\mathcal{E}$, we know that $t = s_1' \; s_2$ where $s_1 \longrightarrow s_1'$. By induction, we may then conclude that

  $$\emptyset \vdash_r s_1' \;:\; x{:}T_1 \to T_2$$

  and rule [T-App] shows that

  $$\emptyset \vdash_r t \;:\; T_2[x := s_2]$$

- s_1 is a value, but s_2 is not a value: Since evaluation is strict, we know that $t = s_1 \; s_2'$ where $s_2 \longrightarrow s_2'$ By induction,

  $$\emptyset \vdash_r s_2' \;:\; T_1$$

  and rule [T-App] then shows that

  $$\emptyset \vdash_r s_1 \; s_2' \;:\; T_2[x := s_2']$$

  From Lemma 10 (Preservation of Subtyping). we know that

  $$\emptyset \vdash T_2[x := s_2'] <: T_2[x := s_2]$$

  Rule [T-Sub] then shows that

  $$\emptyset \vdash_r t \;:\; T_2[x := s_2]$$

- $s_1 = \lambda x : S_1.\; t_1$: In this case, the evaluation rule is [E-App]. Therefore, $x : S_1 \vdash_r t_1 \;:\; T_2$. Lemma 20 then shows that

  $$\emptyset \vdash T_1 <: S_1$$

  and rule [T-Sub] can be used to derive

  $$\emptyset \vdash_r s_2 \;:\; S_1$$

  Lemma 7 (Substitution) then concludes that

  $$\emptyset \vdash_r t_1[x := s_2] \;:\; T_2[x := s_2]$$

- s is a constant other than `cast` applied to its full arity: We examine the `if` constant, assuming the condition is true (the false case is analogous). Recall the type of `if` is

  $$X{:}\texttt{*} \to b{:}\texttt{Bool} \to (\{u{:}\texttt{Unit} \mid b\} \to X) \to (\{u{:}\texttt{Unit} \mid \texttt{not}\ b\} \to X) \to X$$

  We can then assume that

  $$s_1 = \texttt{if}_T \; \texttt{true then} \; v_1 \; \texttt{else} \; v_2$$

  where v_1 and v_2 are values, because of Lemma 18 (Canonical Shapes) and our assumption that the test is true. Since rule [E-IfTrue] must be used,

  $$t = v_1 \; \texttt{unit}$$

  where $\emptyset \vdash_r v_1 \;:\; (u{:}\texttt{Unit.true} \to T)$. Property 6 (Faithfulness) then shows that

  $$\emptyset \models \texttt{true}$$

  and we may use rule [T-App] to conclude

  $$\emptyset \vdash_r v_1 \; \texttt{unit} \;:\; T$$

  Other reduction rules for non-cast constants are as straightforward.

- $s = \langle T \rangle \; s_2$, where s_2 is a value: The values possible for T, which must have type `*`, are given by canonical forms:

  - $T \in \{\texttt{*}, \texttt{Int}, \texttt{Bool}, \texttt{Unit}\}$: A simple inspection is sufficient to show that if the cast succeeds then s_2 can be assigned type T.
  - $T = x{:}T_1 \to T_2$: In this case, rule [E-Cast-Fn] requires that

    $$D = \mathit{domain}(s_2)$$
    $$t = \lambda x : T_1. \langle T_2 \rangle \; (s_2 \; (\langle D \rangle \; x))$$

    We can then easily derive

    $$\emptyset \vdash_r t \;:\; x{:}T_1 \to T_2$$

  - $T = \texttt{Refine}\; T_1 \; f$: We assume the cast doesn't fail (the failure case is vacuous). Thus,

    $$f(\langle T_1 \rangle \; s_2) \longrightarrow^* \texttt{true}$$
    $$\emptyset \vdash_r T_1 \;:\; \texttt{*}$$

    Rule [E-Cast-Refine] then allows us to conclude

    $$t = \langle T_1 \rangle \; s_2$$

We can then easily derive (by [T-Cast] followed by [T-App])

$$\emptyset \vdash_r t : T_1$$

Property 6 (Faithfulness) then shows that

$$\emptyset \models f\ t$$

which allows us to conclude via rule [T-Refine] that

$$\emptyset \vdash_r t : \texttt{Refine}\ T_1\ f$$

- [T-Refine]: In this case,

$$T = \{y\!:\!S \mid p\}$$
$$\emptyset \vdash_r s : S$$
$$\emptyset \vdash_r T : *$$
$$\emptyset \models p[y := s]$$

By induction,

$$\emptyset \vdash_r t : S$$

Property 6 (Preservation) then shows that

$$\emptyset \models p[y := t]$$

and we can use rule [T-Refine] to conclude

$$\emptyset \vdash_r t : T$$

- [T-Let]: Similar to the case for [T-App].

□

E. Soundness of Hybrid Type Checking

The previous sections are sufficient to now prove the main soundness theorem. We divide the proof into two parts. The first shows that our algorithmic subtyping algorithm defined in Figure 9 is a conservation approximation of the subtyping judgment defined in Figure 7. The second part shows the soundness of our cast insertion algorithm specified in Figure 8.

E.1 Conservativity of Algorithmic Subtyping

As specified in Section 5, we must show the following:

Restatement of Lemma 1 (Subtype Algorithm)

1. *If $E \vdash^{\surd}_{alg} S <: T$ then $E \vdash S <: T$.*
2. *If $E \vdash^{\times}_{alg} S <: T$ then $\forall E', S', T'$ that are obtained from E, S, T by replacing the type* `Dynamic` *by any type, we have that $E' \not\vdash S' <: T'$.*

Proof: This follows immediately from Lemmas 21 and 22 below. □

Lemma 21. *If $\vdash E$ and $E \vdash^{\surd}_{alg} S <: T$ then $E \vdash S <: T$.*

Proof: By induction on the derivation $E \vdash^{\surd}_{alg} S <: T$, and case analysis of the last rule used.

- [AS-Dyn-L]: Trivial, since this rule can only conclude "?".
- [AS-Refl], [AS-Dyn-R], [AS-Fun], [AS-Ref-L], [AS-Ref-R], [AS-Var]: These cases follow immediately, or from direct application of the inductive hypothesis.
- [AS-Eval-L]: In this case,

$$S = D[s]$$
$$s \longrightarrow s'$$
$$E \vdash^{\surd}_{alg} D[s'] <: T$$

Since, syntactically, $\mathcal{C} = D$, we have that $D[s] = \mathcal{C}[s]$. This allows us to conclude that $E \vdash S <: T$ via rule [S-Eval].
- [AS-Eval-R]: Similar to the previous case.

□

Lemma 22. *If $\vdash E$ $E \vdash^{\times}_{alg} S <: T$ then $\forall E', S', T'$ that are obtained from E, S, T by replacing the type* `Dynamic` *by any type, we have that $E' \not\vdash S' <: T'$.*

Proof: We proceed by induction on the derivation $E \vdash^{\times}_{alg} S <: T$, and case analysis of the last rule used. Let

$$Dyn(T) \;=\; \{T' \mid T' \text{ is obtained from } T \text{ by replacing each occurrence of } \texttt{Dynamic} \text{ by any type }\}$$

- [AS-Dyn-R], [AS-Refl], [AS-Dyn-L], [AS-Ref-L]: These cases cannot happen, since these rules cannot conclude "$\times$".
- [AS-Fun]: In this case,

  $S = x\!:\!S_1 \to S_2$
  $T = x\!:\!T_1 \to T_2$

  and one of the follow holds:

  $E \vdash^{\times}_{alg} T_1 <: S_1$
  $E, x : T_1 \vdash^{\times}_{alg} S_2 <: T_2$

  We assume the first holds (the other case is similar). By induction,

  $\forall S_1' \in Dyn(S_1).\ \forall T_1' \in Dyn(T_1).\ E \not\vdash T_1' <: S_1'$

  By contrapositive of Lemma 15 (Inversion of function subtyping), which contains the conclusion "If $E \not\vdash T_1' <: S_1'$ then $E \not\vdash x\!:\!S_1' \to S_2' <: x\!:\!T_1' \to T_2'$", we may conclude the following:

  $\forall S' \in Dyn(S).\ \forall T' \in Dyn(T).\ E \not\vdash S' <: T'$

- [AS-Ref-R]: In this case, $T = \{x\!:\!T_1 \,|\, p\}$, and one of the following holds:

  $E, x : S \models^{\times}_{alg} p$
  $E \vdash^{\times}_{alg} S <: T_1$

  We assume the first holds (the other case is similar to the case for [AS-Fun]. By Property 6 (Conservativity), we know that

  $\forall S' \in Dyn(S).\ \forall p' \in Dyn(p).\ E, x : S' \not\models p$

  By the contrapositive of Lemma 16 (Inversion of refinement subtyping), we may conclude

  $\forall S' \in Dyn(S).\ \forall T' \in Dyn(T).\ E \not\vdash S' <: T'$

- [AS-Var]: In this case,

  $E = E_1, x = v : U, E_2$
  $E_1, E_2[x := v] \vdash^{\times}_{alg} S[x := v] <: T[x := v]$

  By induction,

  $\forall S' \in Dyn(S[x := v']).\ \forall T' \in Dyn(T[x := v']).\ E \not\vdash S' <: T'$

  We only are concerned with those S' and T' that can be rewritten as $S''[x := v']$ and $T''[x := v']$ for some $v' \in Dyn(v)$, i.e. those where x is replaced by the same value. Then for any environment containing $x = v' : S$ we have by the contrapositive of Lemma 7 (Substitution) that $E_1', x = v' : S, E_2' \not\vdash S'' <: T''$ where $E_1', x = v' : S, E_2'$ ranges over all of $Dyn(E)$ and S'' and T'' are also universally quantified.
- [AS-Eval-L]: In this case,

  $S = D_S[s]$
  $s \longrightarrow s'$
  $E \vdash^{\times}_{alg} D_S[s'] <: T$

  By induction,

  $\forall S' \in Dyn(D_S[s']).\ \forall T' \in Dyn(T).\ E \not\vdash S' <: T'$

  By the contrapositive of Corollary 12 (Preservation of subtyping under evaluation), we may conclude

  $\forall S' \in Dyn(S).\ \forall T' \in Dyn(T).\ E \not\vdash S' <: T'$

- [AS-Eval-R]: This is similar to the previous case.

□

E.2 Well-typedness of Cast Insertion

Finally, we show that cast insertion produces only well-typed programs. Thus, the only way for a compiled program to fail is to encounter a failed cast.

Restatement of Theorem 5 (Soundness of Hybrid Type Checking)

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

PROOF: This proof follows by simultaneous induction on structure of the derivation of both $E \vdash s \hookrightarrow t : T$ and $E \vdash s \hookrightarrow t \downarrow T$. We proceed by case analysis of the last rule of derivation.

- [C-VAR]: The last step in the derivation uses the following rule:

$$\frac{(x : T) \in E \text{ or } (x : T = t) \in E}{E \vdash x \hookrightarrow x : \{y{:}T \mid y = x\}}$$

  Therefore, one of the following holds:

  $(x : T) \in E$
  $(x : T = t) \in E$

  It then follows that $E \vdash x : T$ by rule [C-VAR].

- [C-CONST]: The last step in the derivation uses the following rule:

$$\frac{}{E \vdash c \hookrightarrow c : ty(c)}$$

  By [T-CONST], $E \vdash c : ty(c)$.

- [C-FUN]: The last step in the derivation uses the following rule:

$$\frac{E \vdash S \hookrightarrow S' \downarrow * \qquad E, x : S' \vdash t \hookrightarrow t' : T}{E \vdash (\lambda x{:}S.\ t) \hookrightarrow (\lambda x{:}S'.\ t') : (x{:}S' \to T)}$$

  In order to show (via rule [T-FUN]) that $E \vdash (\lambda x : S'.t') : (x : S' \to T)$, we must show that

  $E \vdash S' : *$
  $E, x : S' \vdash t' : T$

  These follow from the hypotheses of [C-FUN] and the induction hypothesis. Therefore, $E \vdash (\lambda x : S'.t') : (x : S' \to T)$.

- [C-ARROW]: The last step in the derivation uses the following rule:

$$\frac{E \vdash S \hookrightarrow S' \downarrow * \quad E, x : S' \vdash T \hookrightarrow T' \downarrow *}{E \vdash (x{:}S \to T) \hookrightarrow (x{:}S' \to T') : *}$$

  The proof follows as in the previous case.

- [C-APP1]: The last step in the derivation uses the following rule:

$$\frac{\begin{array}{c}E \vdash t_1 \hookrightarrow t_1' : U \qquad \mathit{unrefine}(U) = x{:}S \to T \\ E \vdash t_2 \hookrightarrow t_2' \downarrow S\end{array}}{E \vdash t_1\ t_2 \hookrightarrow t_1'\ t_2' : T'}$$

  In order to show (via rule [T-APP]) that $E \vdash t_1'\ t_2' : T[x := t_2']$, we must show that

  $E \vdash t_1' : (x : S \to T)$
  $E \vdash t_2' : S$

  By induction, it follows that $E \vdash t_1' : U$. We may then use Lemma 14 to conclude that $E \vdash U <: \mathit{unrefine}(U)$. Using this fact, rule [T-SUB] concludes that $E \vdash t_1' : (x : S \to T)$. By the induction hypothesis, we know that $E \vdash t_2' : S$. Thus, we may apply rule [T-APP] to conclude that $E \vdash t_1'\ t_2' : T[x := t_2']$ holds.

- [C-APP2]: The last step in the derivation uses the following rule:

$$\frac{\begin{array}{c}E \vdash t_1 \hookrightarrow t_1' \downarrow (\texttt{Dynamic} \to \texttt{Dynamic}) \\ E \vdash t_2 \hookrightarrow t_2' \downarrow \texttt{Dynamic}\end{array}}{E \vdash t_1\ t_2 \hookrightarrow t_1'\ t_2' : \texttt{Dynamic}}$$

  In order to show (via rule [T-APP]) that $E \vdash t_1'\ t_2' : \texttt{Dynamic}$ holds, we must show that

  $E \vdash t_1' : (\texttt{Dynamic} \to \texttt{Dynamic})$
  $E \vdash t_2' : \texttt{Dynamic}$

  The two premises of rule [C-APP2], and the induction hypothesis, are sufficient to conclude that these two statements hold.

- [C-LET]: The last step in the derivation uses the following rule:

$$\frac{\begin{array}{c}E \vdash S \hookrightarrow S' \downarrow * \qquad E \vdash v \hookrightarrow v' \downarrow S' \\ E, (x : S' = v') \vdash t \hookrightarrow t' : T \qquad T' = T[x := v']\end{array}}{E \vdash \texttt{let}\ x = v : S\ \texttt{in}\ t \hookrightarrow \texttt{let}\ x = v' : S'\ \texttt{in}\ t' : T'}$$

  In order to show (via rule [T-LET]) that $E \vdash \texttt{let}\ x = v' : S'\ \texttt{in}\ t' : T[x := v']$ holds, we must show that

  $E \vdash v' : S'$
  $E, (x : S' = v') \vdash t' : T$

These follow from the premises of rule [C-Let] and the inductive hypothesis.

- [CC-Ok]: The last step in the derivation uses the following rule:

$$\frac{E \vdash t \hookrightarrow t' : S \qquad E \vdash^{\surd}_{alg} S <: T}{E \vdash t \hookrightarrow t' \downarrow T}$$

In order to show (via rule [T-Sub]) that $E \vdash t' : T$ holds, we must show that both

$E \vdash t' : S$
$E \vdash S <: T$

The first premise of rule [CC-Ok] and the inductive hypothesis imply $E \vdash t' : S$. By Lemma 1 and the assumption that $E \vdash^{\surd}_{alg} S <: T$ imply that $E \vdash S <: T$. Hence we may conclude that $E \vdash t' : T$ by the rule [T-Sub].

- [CC-Chk]: The last step in the derivation uses the following rule:

$$\frac{E \vdash t \hookrightarrow t' : S \qquad E \vdash^{?}_{alg} S <: T}{E \vdash t \hookrightarrow (\langle T \rangle\ t') \downarrow T}$$

In order to show (via rule [T-Sub]) that $E \vdash (\langle T \rangle\ t') : T$ holds, we must show that for some U¡

$E \vdash \langle T \rangle : x : U \to T$
$E \vdash t' : U$

In this case, U, the domain of the cast function $\langle T \rangle$, is `Dynamic`.

The first premise of rule [CC-Chk] and the inductive hypothesis imply that $E \vdash t' : S$. In addition, rule [S-Dyn] implies that $E \vdash S <:$ `Dynamic`. We may then use rule [T-Sub] to conclude that $E \vdash t' :$ `Dynamic`. The type of the cast function $\langle T \rangle$, also known as *cast* T, is `Dynamic` $\to T$. Using rule [T-App], we can then conclude that $E \vdash (\langle T \rangle\ t') : T$.

□