SAGE: Hybrid Checking for Flexible Specifications

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

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

Abstract

Software systems typically contain large APIs that are informally specified and hence easily misused. This paper presents the SAGE programming language, which is designed to enforce precise interface specifications in a flexible manner. The SAGE type system uses a synthesis of the type `Dynamic`, first-class types, and arbitrary refinement types. Since type checking for this expressive language is not statically decidable, SAGE uses *hybrid type checking*, which extends static type checking with dynamic contract checking, automatic theorem proving, and a database of refuted subtype judgments.

1. Introduction

Constructing a large, reliable software system is extremely challenging, due to the difficulty of understanding the system in its entirety. A necessary strategy for controlling this conceptual complexity is to divide the system into modules that communicate via clearly specified interfaces.

The precision of these interface specifications may naturally and appropriately evolve during the course of software development. To illustrate this potential variation, consider the following specifications for the argument to a function `invertMatrix`:

1. The argument can be any (dynamically-typed) value.
2. The argument must be an array of arrays of numbers.
3. The argument must be a *matrix*, that is, a rectangular (non-ragged) array of arrays of numbers.
4. The argument must be a square matrix.
5. The argument must be a square matrix that satisfies the predicate `isInvertible`.

All of these specifications are valid constraints on the argument to `invertMatrix`, although some are obviously more precise than others. Different specifications may be appropriate at different stages of the development process. Simpler specifications facilitate rapid prototyping, whereas more precise specifications provide more correctness guarantees and better documentation.

Traditional statically-typed languages, such as Java, C#, and ML, primarily support the second of these specifications. Dynamically-typed languages such as Scheme primarily support the first specification. Contracts [32, 13, 26, 21, 24, 28, 37, 25, 12, 8] provide a means to document and enforce all of these specifications, but violations are only detected dynamically, resulting in incomplete and late (possibly post-deployment) detection of defects. This paper presents the SAGE programming language and type system, which is designed to support and enforce a wide range of specification methodologies. SAGE verifies correctness properties and detects defects via static checking wherever possible. However, SAGE can also enforce specifications dynamically, when necessary.

On a technical level, the SAGE type system can be viewed as a synthesis of three concepts: the type `Dynamic`; arbitrary refinement types; and first-class types. These features add expressive power in three orthogonal directions, yet they all cooperate neatly within SAGE's hybrid static/dynamic checking framework.

Type `Dynamic`. The type `Dynamic` [23, 1, 39] enables SAGE to support dynamically-typed programming. `Dynamic` is a supertype of all types; any value can be upcast to type `Dynamic`, and a value of declared type `Dynamic` can be implicitly downcast (via a run-time check) to a more precise type. Such downcasts are implicitly inserted when necessary, such as when the operation `add1` (which expects an `Int`) is applied to a variable of type `Dynamic`. Thus, declaring variables to have type `Dynamic` (which is the default if type annotations are omitted) leads to a dynamically-typed, Scheme-like style of programming.

These dynamically-typed programs can later be annotated with traditional type specifications like `Int` and `Bool`. One nice aspect of our system is that the programmer need not fully annotate the program with types in order to reap some benefit. Types enable SAGE to check more properties statically, but it is still able to fall back to dynamic checking whenever the type `Dynamic` is encountered.

Refinement Types. For increased precision, SAGE also supports *refinement types*. For example, the following code snippet defines the type of integers in the range from `lo` (inclusive) to `hi` (exclusive):

```
{ x:Int | lo <= x && x < hi }
```

SAGE extends prior work on decidable refinement types [44, 43, 18, 30, 35] to support arbitrary executable refinement predicates — any boolean expression can be used as a refinement predicate.

First-Class Types. Finally, SAGE elevates types to be first-class values, in the tradition of Pure Type Systems [5]. Thus, types can be returned from functions, which permits function abstractions to abstract over types as well as terms. For example, the following function `Range` takes two integers and returns the *type* of integers within that range:

```
let Range (lo:Int) (hi:Int) : *
    = { x:Int | lo <= x && x < hi };
```

Here, `*` is the type of types and indicates that `Range` returns a type. Similarly, we can pass types to functions, as in the following polymorphic identity function:

Presented at the Scheme and Functional Programming Workshop, September, 2006.

```
let id (T:*) (x:T) : T = x;
```

The traditional limitation of both first-class types and unrestricted refinement types is that they are not statically decidable. SAGE circumvents this difficulty by replacing *static* type checking with *hybrid* type checking [14]. SAGE checks correctness properties and detects defects statically, whenever possible. However, it resorts to dynamic checking for particularly complicated specifications. The overall result is that precise specifications can be enforced, with most errors detected at compile time, and violations of some complicated specifications detected at run time.

1.1 Hybrid Type Checking

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

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

Suppose `factorial` has type $\texttt{Pos} \rightarrow \texttt{Pos}$, where $\texttt{Pos} = \{\texttt{x} : \texttt{Int} \mid \texttt{x} > \texttt{0}\}$ is the type of positive integers, and that t has some type T. If the type checker can prove (or refute) that $T <: \texttt{Pos}$, then this application is well-typed (or ill-typed, respectively). However, if T is a complex refinement type or a complex type-producing computation, the SAGE compiler may be unable to either prove or refute that $T <: \texttt{Pos}$ because subtyping is undecidable.

In this situation, *statically accepting* the application may result in the specification of `factorial` being violated at run time, which is clearly unacceptable. Alternatively, *statically rejecting* such programs would cause the compiler to reject some well-typed programs, as in the Cayenne compiler [4]. Our previous experience with ESC/Java [17] indicates that this is too brittle in practice.

One solution to this dilemma is to require that the programmer provide a proof that $T <: \texttt{Pos}$ (see *e.g.* [7, 36, 10]). While this approach is promising for critical software, it may be somewhat heavyweight for widespread use.

Instead, SAGE enforces the specification of `factorial` dynamically, by inserting the following *type cast* to ensure at run time that the result of t is a positive integer:

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

This approach works regardless of whether T is the type `Dynamic`, a complex refinement type, or a complex type-producing computation.

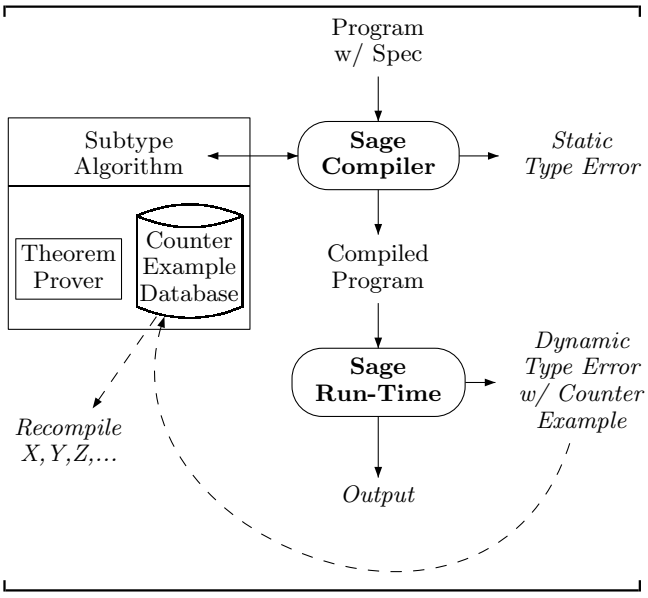
Hybrid combinations of static and dynamic checking are not new. For example, many existing language implementations enforce type safety for arrays using both static type checks and dynamic bounds checks. Hybrid type checking extends this approach to enforce user-defined specifications. Hybrid type checking also extends ideas from soft typing [29, 42, 3], to detect type errors at compile time, in the spirit of static type systems.

Prior work explored hybrid type checking in an idealized setting, that of the simply-typed lambda-calculus with refinements only on the base types `Int` and `Bool` [14]. This paper adapts hybrid type checking to the more technically challenging domain of a rich language that includes all of the features described above. We also provide an implementation and experimental validation of this approach.

1.2 The SAGE Compilation Algorithm

The overall architecture of the SAGE compiler is shown in Figure 1. A key component of the SAGE compiler is its subtype algorithm, which attempts to prove or disprove a subtype relationship $S <: T$ (in the context of an environment E). To obtain adequate precision, the subtype algorithm incorporates the following two modules:

Figure 1: Sage Architecture



Theorem Prover. Testing subtyping between refinement types reduces to testing implication between refinement predicates. In order to reason about these predicates, the subtype algorithm translates each implication between SAGE predicates into a validity test of a logical formula, which can be passed to an automated theorem prover (currently Simplify [11]).

Counter-Example Database. If a compiler-inserted cast from type S to type T fails at run time, SAGE stores in a counter-example database the fact that S is not a subtype of T. The subtype algorithm consults this database during compilation and will subsequently reject any program that relies on S being a subtype of T. Thus, dynamic type errors 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 (or a sufficiently similar) cast, since these programs may also fail at run time. Thus, the counter-example database functions somewhat like a regression test suite, in that it can detect errors in previously compiled programs.

Over time, we predict that the database will grow to be a valuable repository of common but invalid subtype relationships, leading to further improvements in the checker's precision and less reliance on compiler-inserted casts.

The combination of these features yields a subtype algorithm that is quite precise — the number of compiler-inserted casts is very small or zero on all of our benchmarks. Dynamic checks are only necessary for a few particularly complicated cases.

1.3 Contributions

The primary contributions of this paper are as follows:

- We present the SAGE programming language, which supports flexible specifications in a syntactically lightweight manner by combining arbitrary refinement types with first-class types and the type `Dynamic`.

- We present a hybrid type checking algorithm for SAGE that circumvents the decidability limitations of this expressive type language. This type checker accepts all (arbitrarily complicated) well-typed programs, and enforces all interface specifications, either statically or dynamically. The checker integrates compile-time evaluation, theorem proving, and a database of failed type casts.
- We provide experimental results for a prototype implementation of SAGE. These results show that SAGE verifies the vast majority of specifications in our benchmark programs statically.

The following section illustrates the SAGE language through a series of examples. Sections 3 and 4 define the syntax, semantics, and type system for SAGE. Section 5 presents a hybrid compilation 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. We focus primarily on programs with fairly complete specifications to highlight these features. Programs could rely more on the type `Dynamic` than these, albeit with fewer static guarantees.

2.1 Binary Search Trees

We begin with the commonly-studied example of binary search trees, whose SAGE implementation is shown in Figure 2. The variable `Range` is of type `Int` → `Int` → `*`, where `*` is the type of types. Given two integers `lo` and `hi`, the application `Range lo hi` returns the following refinement type describing integers in the range [`lo`, `hi`):

```
{x:Int | lo <= x && x < 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. Thus, no run-time checking is required for this example.

The precise type specifications enable SAGE to detect various common programming errors. For example, suppose we inadvertently used the wrong conditional test:

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

For this (incorrect and ill-typed) program, the SAGE compiler will report that the specification for `insert` is violated by the first recursive call:

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

Similarly, if one of the arguments to the constructor `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)
```

Notably, a traditional type system that does not support precise specifications would not detect either of 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 find (t:PosBST) (x:Range 1 MAXINT) : Bool =
        search 1 MAXINT t x;

let t : PosBST = add (add (add nil 1) 3) 5;
```

Note that this fully-typed BST implementation interoperates with dynamically-typed client code:

```
let t : Dynamic = (add nil 1) in find t 5;
```

2.2 Regular Expressions

We now consider a more complicated specification. Figure 3 declares the `Regexp` data type 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 operators. As an example, the regular expression

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};
```

"`[a-zA-Z0-9]*`" would be represented in our datatype 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`. However, proving that this code is ill-typed 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.

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

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

At run time, these casts check that `username` and `password` are alphanumeric strings satisfying 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`). 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`. These programs may also fail at run time and Sage will also report that they must be recompiled or modified to be accepted by the more-informed checker. It remains to be seen how to best incorporate this feature into a development process.

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). In Sage, we can assign to `printf` the precise type:

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

where the user-defined function

```
Printf_Args : String -> *
```

returns the `printf` argument types for the given format string. For example, `(Printf_Args "%d%d")` evaluates to the type `Int` → `Int` → `Unit`. Calls to `printf` are assigned precise types, such as:

```
printf "%d%d" : Int -> Int -> Unit
```

since this term has type `(Printf_Args "%d%d")`, which in turn evaluates to `Int` → `Int` → `Unit`.

Thus, the Sage language is sufficiently expressive to need no special support for accommodating `printf` and catching errors in `printf` clients statically. In contrast, other language implementations require special rules in the compiler or run time to ensure the type safety of calls to `printf`. For example, Scheme [40] and GHC [19, 38] leave all type checking of arguments to the run time. OCaml [27], on the other hand, performs static checking, but it requires the format string to be constant.

Sage can statically check many uses of `printf` with non-constant format strings, as illustrated by 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 `Printf_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 bounded number of evaluation steps before resorting to dynamic checking. Thus, the following call requires a run-time check:

Figure 4: Syntax, Constants, and Shorthands

Syntax:

$s, t, S, T ::=$	*Terms:*
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:

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

Shorthands:

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

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

As expected, the inserted dynamic cast catches the error.

Our current Sage implementation is not yet able to statically verify that the implementation of `printf` matches its specification (`format:String` → (`Printf_Args format`)). As a result, the compiler inserts a single dynamic type cast into the `printf` implementation. This example illustrates the flexibility of hybrid checking — the `printf` specification is enforced dynamically on the `printf` implementation, but also enforced (primarily) statically on client code. We revisit this example in Section 5.3 to illustrate Sage's compilation algorithm.

3. Language

3.1 Syntax and Informal Semantics

Sage programs are desugared into a small core language, whose syntax and semantics are described in this section. Since types are first class, Sage merges the syntactic categories of terms and types [5]. The syntax of the resulting type/term language is summarized in Figure 4. We use the following naming convention to distinguish the intended use of meta-variables: x, y, z range over regular program variables; X, Y, Z range over type variables; s, t range over regular program terms; and S, T range over types.

The core Sage language includes variables, constants, functions, function applications, and let expressions. The language also includes dependent function types, for which we use Cayenne's [4] 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.

The Sage type system assigns a type to each well-formed term. Since each type is simply a particular kind of term, it is also assigned a type, specifically the type "`*`", which is the type of types [9]. Thus, `Int`, `Bool`, and `Unit` all have type `*`. Also, `*` itself has type `*`.

The unification of types and terms allows us to pass types to and from functions. For example, the following function `UnaryOp` is a type operator that, given a type such as `Int`, returns the type of functions from `Int` to `Int`.

$$\texttt{UnaryOp} \stackrel{\text{def}}{=} \lambda X{:}\texttt{*}.\ (X \to X)$$

Type-valued arguments also support the definition of polymorphic functions, such as `applyTwice`, where the application `applyTwice Int add1` returns a function that adds two to any integer. Thus, polymorphic instantiation is explicit in Sage.

$$\texttt{applyTwice} \stackrel{\text{def}}{=} \lambda X{:}\texttt{*}.\ \lambda f{:}(\texttt{UnaryOp}\ X).\ \lambda x{:}X.\ f(f(x))$$

The constant `Refine` enables precise refinements of existing types. Suppose $f : T \to \texttt{Bool}$ is some arbitrary predicate over type T. Then the type `Refine` T f denotes the *refinement* of T containing all values of type T that satisfy the predicate f. Following Ou et al. [35], we use the shorthand $\{x{:}T \mid t\}$ to abbreviate `Refine` T $(\lambda x{:}T.\ t)$. Thus, $\{x{:}\texttt{Int} \mid x \geq 0\}$ denotes the type of natural numbers.

We use refinement types to assign precise types to constants. For example, as shown in Figure 4, an integer n has the precise type $\{m{:}\texttt{Int} \mid m = n\}$ denoting the singleton set $\{n\}$. Similarly, the type of the operation + specifies that its result is the sum of its arguments:

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

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

The type of the primitive `if` is also described via refinements. 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 constant `fix` enables the definition of recursive functions and recursive types. For example, the type of integer lists is defined via the least fixpoint operation:

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

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

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

Figure 5: Evaluation Rules

Evaluation $\boxed{s \longrightarrow t}$

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

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

$$\texttt{let } x = v : S \texttt{ in } t \longrightarrow t[x := v] \qquad \text{[E-Let]}$$

$$\texttt{not true} \longrightarrow \texttt{false} \qquad \text{[E-Not1]}$$

$$\texttt{not false} \longrightarrow \texttt{true} \qquad \text{[E-Not2]}$$

$$\texttt{if}_T\ \texttt{true}\ v_1\ v_2 \longrightarrow v_1\ \texttt{unit} \qquad \text{[E-If1]}$$

$$\texttt{if}_T\ \texttt{false}\ v_1\ v_2 \longrightarrow v_2\ \texttt{unit} \qquad \text{[E-If2]}$$

$$+\ n_1\ n_2 \longrightarrow n \quad n = (n_1 + n_2) \qquad \text{[E-Add]}$$

$$=\ v_1\ v_2 \longrightarrow c \quad c = (v_1 \equiv v_2) \qquad \text{[E-Eq]}$$

$$\langle \texttt{Bool} \rangle\ \texttt{true} \longrightarrow \texttt{true} \qquad \text{[E-Cast-Bool1]}$$

$$\langle \texttt{Bool} \rangle\ \texttt{false} \longrightarrow \texttt{false} \qquad \text{[E-Cast-Bool2]}$$

$$\langle \texttt{Unit} \rangle\ \texttt{unit} \longrightarrow \texttt{unit} \qquad \text{[E-Cast-Unit]}$$

$$\langle \texttt{Int} \rangle\ n \longrightarrow n \qquad \text{[E-Cast-Int]}$$

$$\langle \texttt{Dynamic} \rangle\ v \longrightarrow v \qquad \text{[E-Cast-Dyn]}$$

$$\langle x{:}S \to T \rangle\ v \longrightarrow \lambda x{:}S.\ \langle T \rangle\ (v\ (\langle D \rangle\ x)) \quad \text{where } D = \mathit{domain}(v) \qquad \text{[E-Cast-Fn]}$$

$$\langle \texttt{Refine } T\ f \rangle\ v \longrightarrow \langle T \rangle\ v \quad \text{if } f\ (\langle T \rangle\ v) \longrightarrow^* \texttt{true} \qquad \text{[E-Refine]}$$

$$\langle * \rangle\ v \longrightarrow v \quad \text{if } v \in \{\texttt{Int}, \texttt{Bool}, \texttt{Unit}, \texttt{Dynamic}, x{:}S \to T, \texttt{fix} * f\ \} \qquad \text{[E-Cast-Type]}$$

$$\mathcal{S}[\texttt{fix}\ U\ v] \longrightarrow \mathcal{S}[v\ (\texttt{fix}\ U\ v)] \qquad \text{[E-Fix]}$$

$$\mathcal{E} ::= \bullet \mid \mathcal{E}\ t \mid v\ \mathcal{E} \qquad \textit{Evaluation Contexts}$$

$$\mathcal{S} ::= \bullet \mid v \mid \langle \bullet \rangle\ v \qquad \textit{Strict Contexts}$$

$$u, v, U, V ::= \qquad \textit{Values}$$

$\lambda x{:}S.\ t$	abstraction
$x{:}S \to T$	function type
c	constant
$c\ v_1\ \ldots\ v_n$	constant, $0 < n < \mathit{arity}(c)$
$\texttt{Refine}\ U\ v$	refinement
$\texttt{fix}\ U\ v$	recursive type

(Here, `Sum` and `Pair` are the usual type constructors for sums and pairs, respectively.)

The Sage language includes two constants that are crucial for enabling hybrid type checking: `Dynamic` and `cast`. The type constant `Dynamic` [1, 39] can be thought of as the most general type. Every value has type `Dynamic`, and casts can be used to convert values from type `Dynamic` to other types (and of course such downcasts may fail if applied to inappropriate values).

The constant `cast` performs dynamic checks or coercions between types. It takes as arguments a type T and a value (of type `Dynamic`), and it attempts to cast that value to type T. We use the shorthand $\langle T \rangle\ t$ to abbreviate `cast` $T\ t$. Thus, for example, the expression

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

casts the integer y to the refinement type of natural numbers, and fails if y is negative.

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 $\mathcal{E}$. Application, let expressions, and the basic integer and boolean operations behave as expected. Rule [E-Eq] uses syntactic equality ($\equiv$) to test equivalence of all values, including function values[1].

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

Casts to function and refinement types are more complex. First, the following partial function *domain* returns the domain of a function value, and is defined by:

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

The rule [E-Cast-Fn] casts a function v to type $x : S \to T$ by creating a new function:

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

where $D = \mathit{domain}(v)$ is the domain type of the function v. This new function takes a value x of type S, casts it to D, applies the given function v, and casts the result to the desired result type T. Thus, casting a function to a different function type will always succeed, since the domain and range values are checked lazily, in a manner reminiscent of higher-order contracts [13].

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

Casts to type `*` succeed only for special values of type `*`, via the rule [E-Cast-Type].

The operation `fix` is used to define recursive functions and types, which are considered values, and hence $\texttt{fix}\ U\ v$ is also a value. However, when this construct $\texttt{fix}\ U\ v$ appears in a *strict position* (*i.e.*, in a function position or in a cast), the rule [E-fix] performs one step of unrolling to yield $v\ (\texttt{fix}\ U\ v)$.

4. Type System

The Sage type system is defined via the type rules and judgments shown in Figure 6. Typing is performed in an environment E that binds variables to types and, in some cases, to values. We assume that variables are bound at most once in an environment and, as usual, we apply implicit α-renaming of bound variables to maintain this assumption and to ensure that substitutions are capture-avoiding.

The Sage type system guarantees *progress* (*i.e.*, that well-typed programs can only get stuck due to failed casts) and *preservation* (*i.e.*, that evaluation of a term preserves its type). The proofs appear in a companion report [22].

The main typing judgment

$$E \vdash t : T$$

[1] A semantic notion of equality for primitive types could provide additional flexibility, although such a notion would clearly be undecidable for higher-order types. In practice, syntactic equality has been sufficient.

Figure 6: Type Rules

$E ::=$ *Environments:*
- $\emptyset$ — empty environment
- $E, x : T$ — environment extension
- $E, x = v : T$ — environment term extension

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

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

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

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

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

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

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

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

assigns type T to term t in the environment E. In the rule [T-Const], the auxiliary function ty returns the type of the 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 in an extended environment and returns the dependent function type $x{:}S \to T$, where x may occur free in T. The type $x{:}S \to T$ is itself a term, which is assigned type $*$ by rule [T-Arrow], provided that both S and T have type $*$ in appropriate environments.

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 is in the domain of t_1. The result type is T with all occurrences of the formal parameter x replaced by the actual parameter t_2.

The type rule [T-Let] for $\texttt{let}\ x = v : S\ \texttt{in}\ t$ first checks that the type of the bound value v is S. Then t is typed in an environment that contains both the type and the value of x. These precise 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].

The subtype judgment

$$E \vdash S <: T$$

states that S is a subtype of T in the environment E, and it is defined as the least solution to the collection of subtype rules 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] for function types checks the usual contravariant/covariant subtype requirements on function domains and codomains. The rule [S-Var] hygienically replaces a variable with the value to which it is bound.

Figure 7: Subtype Rules

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

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

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

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

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

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

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

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

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

The remaining rules are less conventional. Rules [S-Eval-L] and [S-Eval-R] state that the subtype relation is closed under evaluation of terms in arbitrary positions. In these rules, $\mathcal{C}$ denotes an arbitrary context:

$$\begin{aligned} \mathcal{C} \quad ::= \quad & \bullet \mid \mathcal{C}\ t \mid t\ \mathcal{C} \mid \lambda x{:}\mathcal{C}.\ t \mid \lambda x{:}T.\ \mathcal{C} \\ & \mid \texttt{let}\ x = \mathcal{C} : S\ \texttt{in}\ t \mid \texttt{let}\ x = t : \mathcal{C}\ \texttt{in}\ t \\ & \mid \texttt{let}\ x = t : S\ \texttt{in}\ \mathcal{C} \end{aligned}$$

The rule [S-Ref-L] states that, if S is a subtype of T, then any refinement of S is also a subtype of T. When S is a subtype of T, the rule [S-Ref-R] invokes the theorem proving judgment $E \models f\ x$, discussed below, to determine if $f\ x$ is valid for all values x of type S. If so, then S is a subtype of $\texttt{Refine}\ T\ f$.

Our type system is parameterized with respect to the theorem proving judgment

$$E \models t$$

which defines the validity of term t in an environment E. We specify the interface between the type system and the theorem prover via the following axioms (akin to those found in [35]), which are sufficient to prove soundness of the type system. In the following, all environments are assumed to be well-formed [22].

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 $E, (x : S), F \models t$.

An alternative to these axioms is to define the validity judgment $E \models t$ directly. In such an approach, we could say that a term t is valid if, for all *closing substitutions* σ that map the names in E to terms consistent with their types, the term $\sigma(t)$ evaluates to true:

$$\frac{\begin{array}{c}\forall \sigma \text{ if } \sigma \text{ is consistent with } E \\ \text{then } \sigma(t) \longrightarrow^* \texttt{true}\end{array}}{E \models t} \quad [\textsc{Validity}]$$

This approach has several drawbacks. First, the rule makes the type system less flexible with regard to the underlying logic. More importantly, however, the rule creates a cyclic dependency between validity and the typing of terms in σ. Thus, consistency of the resulting system is non-obvious and remains an open question. For these reasons, we stick to the original axiomatization of theorem proving.

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.

5. Hybrid Type Compilation

The Sage hybrid type checking (or *compilation*) algorithm shown in Figure 8 type checks programs and simultaneously inserts dynamic casts. These casts compensate for inevitable limitations in the Sage subtype algorithm, which is a conservative approximation of the subtype relation.

5.1 Algorithmic Subtyping

For any subtype query $E \vdash S <: T$, the algorithmic subtyping judgment $E \vdash^a_{alg} S <: T$ returns a result $a \in \{\surd, \times, ?\}$ depending on whether the algorithm succeeds in proving ($\surd$) or refuting ($\times$) the subtype query, or whether it cannot decide the query (?). Our algorithm conservatively approximates the subtype specification in Figure 6. However, special care must be taken in the treatment of `Dynamic`. Since we would like values of type `Dynamic` to be implicitly cast to other types, such as `Int`, the subtype algorithm should conclude $E \vdash^?_{alg}$ `Dynamic` $<:$ `Int` (forcing a cast from `Dynamic` to `Int`), even though clearly $E \nvdash$ `Dynamic` $<:$ `Int`. We thus formalize our requirements for the subtype algorithm as the following lemma.

Lemma 1 (Algorithmic Subtyping).

1. *If* $E \vdash^{\surd}_{alg} S <: T$ *then* $E \vdash S <: T$.
2. *If* $E \vdash^{\times}_{alg} T_1 <: T_2$ *then* $\forall F, S_1, S_2$ *that are obtained from* E, T_1, T_2 *by replacing the type* `Dynamic` *by any type, we have that* $F \nvdash S_1 <: S_2$.

Clearly, a naïve subtype algorithm could always return the result "?" and thus trivially satisfy these requirements, but more precise results enable Sage to verify more properties and to detect more errors at compile time.

This specification of the subtype algorithm is sufficient for describing the compilation process, and we defer presenting the full details of the algorithm until Section 6.

Figure 8: Compilation Rules

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

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

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

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

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

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

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

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

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

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

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

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

separate algorithm

5.2 Checking and Compilation

The compilation judgment

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

compiles the source term s, in environment E, to a compiled term t (possibly with additional casts), where T is the type of 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 type T.

Many of the compilation rules are similar to the corresponding type rules, *e.g.*, [C-Var] and [C-Const]. The rule [C-Fun] compiles a function $\lambda x{:}S.\ t$ by compiling S to some type S' of type $*_{\square}$ and then compiling t (in the extended environment $E, x : S'$) to a term t' of type T. The rule returns the compiled function $\lambda x{:}S'.\ t'$ of type $x{:}S' \to T$. The rule [C-Arrow] compiles a function type by compiling the two

component types and checking that they both have type `*`. The rule [C-Let] compiles the term `let` $x = v : S$ `in` t by recursively compiling v, S and t in appropriate environments.

The rules for function application are more interesting. The rule [C-App1] compiles an application $t_1\ t_2$ by compiling the function t_1 to some term t_1' of some type U. The type U may be a function type embedded inside refinements. In order to extract the actual type of the parameter to the function, we use *unrefine* to remove any outer refinements of U before checking the type of the argument t_2 against the expected type. Formally, *unrefine* is defined as follows:

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

The last clause permits S to be simplified via evaluation while removing outer refinements. Given the expressiveness of the type system, this evaluation may not converge within a given time bound. Hence, to ensure that our compiler accepts all (arbitrarily complicated) well-typed programs, the rule [C-App2] provides a backup compilation strategy for applications that requires less static analysis, but performs more dynamic checking. This rule checks that the function expression has the most general function type `Dynamic` $\rightarrow$ `Dynamic`, and correspondingly coerces t_2 to type `Dynamic`, resulting in an application with type `Dynamic`.

The rules defining the compilation and checking judgment

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

illustrate the key ideas of hybrid type checking. The rules [CC-Ok] and [CC-Chk] compile the given term and check that the compiled term has the expected type T via the algorithmic subtyping judgment

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

If this judgment succeeds ($a = \surd$), then [CC-OK] returns the compiled term. If the subtyping judgment is undecided ($a = ?$), then [CC-Chk] encloses the compiled term in the cast $\langle T \rangle$ to preserve dynamic type safety.

The compilation rules guarantee that a compiled program is well-typed [22], and thus compiled programs only go wrong due to failed casts. In addition, this property permits type-directed optimizations on compiled code.

5.3 Example

To illustrate how Sage verifies specifications statically when possible, but dynamically when necessary, we consider the compilation of the following term:

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

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):

$$\emptyset \vdash \texttt{(printf "\%d")} \hookrightarrow \texttt{(printf "\%d") : (Printf\_Args "\%d")}$$

The rule [C-App1] then calls the function *unrefine* to evaluate `(Printf_Args "%d")` to the normal form `Int` $\rightarrow$ `Unit`. Since 4 has type `Int`, the term t is therefore accepted as is; no casts are needed.

However, the computation for `(Printf_Args "%d")` may not terminate within a preset time limit. In this case, the compiler uses the rule [C-App2] to compile t into the code:

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

At run time, `(printf "%d")` will evaluate to some function $(\lambda x{:}\texttt{Int}.\ t')$ that expects an `Int`, yielding the application:

$$(\langle \texttt{Dynamic} \rightarrow \texttt{Dynamic} \rangle\ (\lambda x{:}\texttt{Int}.\ t'))\ \ 4$$

The rule [E-Cast-Fn] then reduces this term to:

$$(\lambda x{:}\texttt{Dynamic}.\ \langle \texttt{Dynamic} \rangle\ ((\lambda x{:}\texttt{Int}.\ t')\ (\langle \texttt{Int} \rangle\ x)))\ \ 4$$

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

6. Implementation

Our prototype Sage implementation consists of roughly 5,000 lines of OCaml code. The run time implements the semantics from Section 3, with one extension for supporting the counter-example database when casts fail. 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, the run time inserts an entry into the database asserting that $E \not\vdash S <: T$.

Function casts must be treated with care to ensure blame is assigned appropriately upon failure [13]. In particular, if a cast inserted during the lazy evaluation of a function cast fails, an entry for the original, top-level function cast is inserted into the database rather than for the "smaller" cast on the argument or return value.

The Sage subtype algorithm computes the least fixed point of the algorithmic rules in Figure 9. These rules return 3-valued results which are combined with the 3-valued conjunction operator $\otimes$:

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

The algorithm attempts to apply the rules in the order in which they are presented in the figure. If no rule applies, the algorithm returns $E \vdash^{?}_{alg} S <: T$. Most of the rules are straightforward, and we focus primarily on the most interesting rules:

[AS-Db]: Before applying any other rules, the algorithm attempts to refute that $E \vdash S <: T$ by querying the database of previously refuted subtype relationships. The judgment $E \vdash^{\times}_{db} S <: T$ indicates that the database includes an entry stating that S is not a subtype of T in an environment E', where E and E' are compatible in the sense that they include the same bindings for the free variables in S and T. This compatibility requirement ensures that we only re-use a refutation in a typing context in which it is meaningful.

[AS-Eval-L] and [AS-Eval-R]: These two rules evaluate the terms representing types. The algorithm only applies these two rules a bounded number of times before timing out and forcing the algorithm to use a different rule or return "?". This prevents non-terminating computation as well as infinite unrolling of recursive types.

[AS-Dyn-L] and [AS-Dyn-R]: These rules ensure that any type can be considered a subtype of `Dynamic` and that converting from `Dynamic` to any type requires an explicit coercion.

[AS-Ref-R]: This rule for checking whether S is a subtype of a specific refinement type relies on a theorem-proving algorithm, $E \models^{a}_{alg} t$, for checking validity. This algorithm is an approximation of some validity judgment $E \models t$ sat-

Figure 9: Subtyping Algorithm

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

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

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

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

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

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

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

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

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

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

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

$D ::= \ \bullet \mid N\ D$ 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

isfying the axioms in Section 4. As with subtyping, the result $a \in \{\surd, ?, \times\}$ indicates whether or not the theorem prover could prove or refute the validity of t. The algorithmic theorem proving judgment must be conservative with respect to the logic it is approximating, as captured in the following requirement:

REQUIREMENT 2 (Algorithmic Theorem Proving).

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 implementation of this theorem-proving algorithm translates the query $E \models^{a}_{alg} t$ into input for the Simplify theorem prover [11]. 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`. Given the incompleteness of Simplify (and other theorem provers), care must be taken in how the Simplify results are interpreted. For example, on the translated version of the query

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

Simplify returns `Invalid`, because it is incomplete for arbitrary multiplication. In this case, the SAGE theorem prover returns the result "?" to indicate that the validity of the query is unknown. We currently assume that the theorem prover 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.

Assuming that $E \models^{a}_{alg} t$ satisfies Requirement 2 and that $E \vdash^{\times}_{db} S <: T$ only if $E \not\vdash S <: T$ (meaning that the database only contains invalid subtype tests), it is straightforward to show that the subtype algorithm $E \vdash^{a}_{alg} S <: T$ satisfies Lemma 1.

7. Experimental Results

We evaluated the SAGE language and implementation using the benchmarks listed in Figure 10. 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 $\surd$), refuted (denoted by $\times$), and left undecided (denoted by ?). The benchmarks are all well-typed, so no subtype queries are refuted. Note that the theorem prover enables SAGE to decide many more subtype queries. In particular, many of the benchmarks include complex refinement types that use integer arithmetic to specify ordering and structure invariants; theorem proving is particularly helpful in verifying these benchmarks.

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 sub-

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

type judgments. Some complicated tests in `stlc.sage` and `printf.sage` must also be checked dynamically.

Despite the use of a theorem prover, compilation times for these benchmarks is quite manageable. On a 3GHz Pentium 4 Xeon processor running Linux 2.6.14, compilation 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.

8. Related Work

The enforcement of complex program specifications, or *contracts*, is the subject of a large body of prior work [32, 13, 26, 21, 24, 28, 37, 25, 12, 8]. Since these contracts are typically not expressible in classical type systems, they have previously been relegated to dynamic checking, as in, for example, Eiffel [32]. Eiffel's 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 [31].

The static checking tool ESC/Java [17] supports expressive JML specifications [26]. 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. In contrast, hybrid type checking only produces error messages for provably ill-typed programs.

The Spec# programming system extends C# with expressive specifications [6], including preconditions, postconditions, and non-null annotations. Specifications are enforced dynamically, and can be also checked statically via a separate tool. The system is somewhat less tightly integrated than in Sage. For example, successful static verification does not automatically remove the corresponding dynamic checks.

Recent work on advanced type systems has influenced our choice of how to express program invariants. In particular, Freeman and Pfenning [18] extended ML with another form of refinement types. They work focuses on providing both decidable type checking and type inference, instead of on supporting arbitrary refinement predicates.

Xi and Pfenning have explored applications of dependent types in Dependent ML [44, 43]. Decidability of type checking is preserved by appropriately restricting which terms can appear in types. Despite these restrictions, a number of interesting examples can be expressed in Dependent ML. Our system of dependent types extends theirs with arbitrary executable refinement predicates, and the hybrid type checking infrastructure is designed to cope with the resulting undecidability. In a complementary approach, Chen and Xi [10] address decidability limitations by providing a mechanism through which the programmer can provide proofs of subtle properties in the source code.

Recently, Ou, Tan, Mandelbaum, and Walker developed a dependent type system that leverages dynamic checks [35] in a way similar to Sage. Unlike Sage, their system is decidable, and they leverage dynamic checks to reduce the need for precise type annotations in explicitly labeled regions of programs. They consider mutable data, which we intend to add to Sage in the future. We are exploring other language features, such as objects [16], as well.

Barendregt introduced the unification of types and terms, which allows types to be flexibly expressed as complex expressions, while simplifying the underlying theory [5]. The language Cayenne adopts this approach and copes with the resulting undecidability of type checking by allowing a maximum number of steps, somewhat like a timeout, before reporting to the user that typing has failed [4]. 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.

Other authors have considered pragmatic combinations of both static and dynamic checking. Abadi, Cardelli, Pierce and Plotkin [1] extended a static type system with a type `Dynamic` that could be explicitly cast to and from any other type (with appropriate run-time checks). Henglein characterized the *completion process* of inserting the necessary coercions, and presented a rewriting system for generating minimal completions [23]. Thatte developed a similar system in which the necessary casts are implicit [39]. For Scheme, soft type systems [29, 42, 3, 15] 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 [33] is a sophisticated hybrid analysis for preventing the ubiquitous array bounds violations in the C programming language. Unlike our proposed approach, it does not detect errors statically. Instead, the static analysis is used to optimize the run-time analysis. Specialized hybrid analyses have been proposed for other problems as well, such as data race condition checking [41, 34, 2].

9. Conclusions and Future Work

Program specifications are essential for modular development of reliable software. Sage uses a synthesis of first-class types, `Dynamic`, and refinement types to enforce precise specifications in a flexible manner. Our hybrid checking algorithm extends traditional type checking with a theorem prover, a database of counter-examples, and the ability to insert dynamic checks when necessary. Experimental results show that Sage can verify many correctness properties at compile time. We believe that Sage illustrates a promising approach for reliable software development.

A number of opportunities remain for future work. The benefits of the refuted subtype database can clearly be amplified by maintaining a single repository for all local and non-local users of Sage. We also plan to integrate randomized or directed [20] testing to refute additional validity queries, thereby detecting more errors at compile time. Since precise type inference for Sage is undecidable, we plan to develop hybrid algorithms that infer precise types

for most type variables, and that may occasionally infer the looser type `Dynamic` in particularly complicated situations.

Acknowledgments: We thank Robby Findler and Bo Adler for useful feedback on this work.

References

[1] 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.

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

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

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

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

[6] M. Barnett, K. R. M. Leino, and W. Schulte. The Spec# programming system: An overview. In *Construction and Analysis of Safe, Secure, and Interoperable Smart Devices: International Workshop*, pages 49–69, 2005.

[7] Y. Bertot and P. Castéran. *Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions*. 2004.

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

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

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

[11] D. L. Detlefs, G. Nelson, and J. B. Saxe. Simplify: A theorem prover for program checking. Research Report HPL-2003-148, HP Labs, 2003.

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

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

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

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

[16] C. Flanagan, S. N. Freund, and A. Tomb. Hybrid object types, specifications, and invariants. In *Workshop on Foundations and Developments of Object-Oriented Languages*, 2006.

[17] 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.

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

[19] The Glasgow Haskell Compiler, release 6.4.1. Available from `http://www.haskell.org/ghc`, 2006.

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

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

[22] J. Gronski, K. Knowles, A. Tomb, S. N. Freund, and C. Flanagan. Sage: Practical hybrid checking for expressive types and specifications, extended report. Available at `http://www.soe.ucsc.edu/~cormac/papers/sage-full.ps`, 2006.

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

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

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

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

[27] X. Leroy (with 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.

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

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

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

[31] 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.

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

[33] 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.

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

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

[36] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In *International Conference on Automated Deduction*, pages 748–752, 1992.

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

[38] Paul Hudak and Simon Peyton-Jones and Philip Wadler (eds.). Report on the programming language Haskell: A non-strict, purely functional language version 1.2. *SIGPLAN Notices*, 27(5), 1992.

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

[40] Sussman, G.J. and G.L. Steele Jr. Scheme: An interpreter for extended lambda calculus. Memo 349, MIT AI Lab, 1975.

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

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

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

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