Semantics and Types for Objects with First-Class Member Names

Joe Gibbs Politz
Brown University
joe@cs.brown.edu

Arjun Guha
Cornell University
arjun@cs.cornell.edu

Shriram Krishnamurthi
Brown University
sk@cs.brown.edu

Abstract

Objects in many programming languages are indexed by first-class strings, not just first-order names. We define λ_S^{ob} ("lambda sob"), an object calculus for such languages, and prove its untyped soundness using Coq. We then develop a type system for λ_S^{ob} that is built around *string pattern types*, which describe (possibly infinite) collections of members. We define subtyping over such types, extend them to handle inheritance, and discuss the relationship between the two. We enrich the type system to recognize tests for whether members are present, and briefly discuss exposed inheritance chains. The resulting language permits the ascription of meaningful types to programs that exploit first-class member names for object-relational mapping, sandboxing, dictionaries, etc. We prove that well-typed programs never signal member-not-found errors, even when they use reflection and first-class member names. We briefly discuss the implementation of these types in a prototype type-checker.

1. Introduction

In most statically-typed object-oriented languages, an object's type or class enumerates its member names and their types. In "scripting languages", member names are first-class strings that can be computed dynamically. In recent years, programmers using these languages have employed first-class member names to create useful abstractions that are applied broadly. Of course, this power also leads to problems, especially when combined with other features like inheritance.

This paper explores uses of first-class member names, a dynamic semantics of their runtime behavior, and a static semantics with a traditional soundness theorem for these untraditional programs. In section 2, we present examples from several languages that highlight uses of first-class members. These examples also show how these languages differ from traditional object calculi. In section 3 we present an object calculus, λ_S^{ob}, which features the essentials of objects in these languages. In section 4, we explore types for λ_S^{ob}. The challenge is to design types that can properly capture the consequences of first-class member names. We especially focus on the treatment of subtyping, inheritance, and their interaction, as well as reflective features such as tests of member presence. Finally, in section 5 we briefly discuss an implementation and its uses.

In summary, we make the following contributions:

1. extract the principle of first-class member names from existing languages;
2. provide a dynamic semantics that distills this feature;
3. identify key problems for type-checking objects in programs that employ first-class member names;
4. extend traditional record typing with sound types to describe objects that use first-class member names; and,
5. briefly discuss a prototype implementation.

We build up our type system incrementally. All elided proofs and definitions are available online:

`http://www.cs.brown.edu/research/plt/dl/fcfn/v1/`

2. Using First-Class Member Names

Most languages with objects, not only scripting languages, allow programmers to use first-class strings to index members. The syntactic overhead differs, as does the prevalence of the feature's use within the corpus of programs in the language. This section explores first-class member names in existing languages, and highlights several of their uses.

2.1 Objects with First-Class Member Names

In Lua and JavaScript, `obj.x` is merely syntactic sugar for `obj["x"]`, so any member can be indexed by a runtime string value:

```
obj = {};
obj["xy" + "z"] = 22;
obj["x" + "yz"]; // evaluates to 22 in both languages
```

Python and Ruby support this pattern with only minor syntactic overhead. In Python:

```
class C(object):
  pass
obj = C()
setattr(obj, "x" + "yz", 22)
getattr(obj, "xy" + "z")  # evaluates to 22
```

and in Ruby:

```
class C; end
obj = C.new
class << obj
  define_method(("x" + "yz").to_sym) do; return 22; end
end
obj.send(("xy" + "z").to_sym) # evaluates to 22
```

In fact, even in Java, programmers are not forced to use first-order labels to refer to member names; it is merely a convenient

default. Java, for example, has `java.lang.Class.getMethod()`, which returns the method with a given name and parameter set.[1]

2.2 Leveraging First-Class Member Names

Once member names are merely strings, programmers can manipulate them as mere data. The input to member lookup and update can come by concatenating strings, from configuration files, from reflected runtime values, via `Math.random()`, etc. This flexibility has been used, quite creatively, in many contexts.

Django The Python Django ORM dynamically builds classes based on database tables. In the following snippet, it adds a member `attr_name`, that represents a database column, to a class `new_class`, which it is constructing on-the-fly:[2]

```
attr_name = '%s_ptr' % base._meta.module_name
field = OneToOneField(base, name=attr_name,
        auto_created=True, parent_link=True)
new_class.add_to_class(attr_name, field)
```

`attr_name` concatenates `"_ptr"` to `base._meta.module_name`. It names a new member that is used in the resulting class as an accessor of another database table. For example, if the `Paper` table referenced the `Submittable` table, `Paper` instances would have a member `submittable_ptr`. Django has a number of pattern-based rules for inserting new members into classes, carefully designed to provide an expressive, object-based API to the client of the ORM. Its implementation, which is in pure Python, requires no extra-lingual metaprogramming tools.

Ruby on Rails When setting up a user-defined model, ActiveRecord iterates over the members of an object and only processes members that match certain patterns:[3]

```
attributes.each do |k, v|
  if k.include?("(")
    multi_parameter_attributes << [ k, v ]
  elsif respond_to?("#{k}=")
    send("#{k}=", v)
  else
    raise(UnknownAttributeError, "unknown attr: #{k}")
  end
end
```

The first pattern, `k.include?("(")`, checks the shape of the member name `k`, and the second pattern checks if the object has a member called `k + "="`. This is a Ruby convention for the setter of an object attribute, so this block of code invokes a setter function for each element in a key-value list. As with Django, ActiveRecord is leveraging first-class member names in order to provide an API implemented in pure Ruby that it couldn't otherwise without richer metaprogramming facilities.

Java Beans Java Beans provide a flexible component-based mechanism for composing applications. The Beans API uses reflective reasoning on canonical naming patterns to construct classes on-the-fly. For example, from `java.beans.Introspector`: "If we don't find explicit BeanInfo on a class, we use low-level reflection to study the methods of the class and apply standard design patterns to identify property accessors, event sources, or public methods."[4] Properties of Beans are not known statically, so the API exposes a `PropertyDescriptor` class that provides methods including `getPropertyType` and `getReadMethod`, which return reflective descriptions of the types of properties of Beans.

```
var banned = { "caller": true, "arguments": true, ... };

function reject_name(name) {
  return ((typeof name !== 'number' || name < 0)
          && (typeof name !== 'string'
             || name.charAt(0) === '_'
             || name.slice(-1) === '_'
             || name.charAt(0) === '-'))
         || banned[name];
}
```

Figure 1. Check for Banned Names in ADsafe

Sandboxes JavaScript sandboxes like ADsafe [10] and Caja [24] use a combination of static and dynamic checks to ensure that untrusted programs do not access *banned fields* that may contain dangerous capabilities. To enforce this dynamically, all member-lookup expressions (`obj[name]`) in untrusted code are rewritten to check whether `name` is banned. Figure 1 is ADsafe's check; it uses a collection of ad hoc tests and also ensures that `name` is not the name of any member in the `banned` object, which is effectively used as a set of names. Caja goes further: it employs eight different patterns for encoding information related to members.[5] For example, the Caja runtime adds a boolean-flagged member named `s + "_w__"` for each member `s` in an object, to denote whether it is writable or not. This lets Caja emulate a number of features that aren't found in its target language, JavaScript.

Objects as Arrays In Lua and JavaScript, built-in operations like splitting and sorting, and simple indexed `for` loops, work with any object that has numeric members. For example, in the following program, JavaScript's built-in `Array.prototype.sort` can be used on `obj`:

```
var obj = {length: 3, 0: 'def', 1: 'abc', 2: 'hij'};
// Array.prototype holds built-in methods
Array.prototype.sort.call(obj);
// evaluates to true
obj[0] == 'abc' && obj[1] == 'def' && obj[2] == 'hij'
```

In fact, the JavaScript specification states that a string P is a valid array index "if and only if ToString(ToUint32(P)) is equal to P and ToUint32(P) is not equal to $2^{32} - 1$."[6]

2.3 The Perils of First-Class Member Names

Along with their great flexibility, first-class member names bring their own set of subtle error cases. First, developers must program defensively against the possibility that dereferenced members are not present (we see an example of this in the Ruby example from the last section, which explicitly uses `respond_to?` to check that the setter is present before using it). These sorts of reflective checks are common in programs that use first-class member names; a summary of such reflective operators across languages is included in figure 2. In other words, first-class member names put developers back in the era before type systems could guard against run-time *member not found* errors. Our type system (section 4) restores them to the happy state of finding these errors statically.

Second, programmers also make mistakes because they sometimes fail to respect that objects truly are *objects*, which means

[1] `http://docs.oracle.com/javase/1.4.2/docs/api/java/lang/Class.html#getMethod(java.lang.String,java.lang.Class[])`

[2] `https://github.com/django/django/blob/master/django/db/models/base.py#L157`

[3] `https://github.com/rails/rails/blob/master/activerecord/lib/active_record/base.rb#L1717`

[4] `http://download.oracle.com/javase/tutorial/javabeans/introspection/index.html`

[5] Personal communication with Jasvir Nagra, technical lead of Google Caja.

[6] `http://es5.github.com/#x15.4`

	Structural	
Ruby	`o.respond_to?(p)`	`o.methods`
Python	`hasattr(o, p)`	`dir(o)`
JavaScript	`o.hasOwnProperty(p)`	`for (x in o)`
Lua	`o[p] == nil`	

	Nominal
Ruby	`o.is_a?(c)`
Python	`isinstance(o, c)`
JavaScript	`o instanceof c`
Lua	`getmetatable(o) == c`

Figure 2. Reflection APIs

they inherit members.[7] When member names are computed at runtime, this can lead to subtle errors. For instance, Google Docs uses an object as a dictionary-like data structure internally, which uses strings from the user's document as keys in the dictionary. Since (in most major browsers) JavaScript exposes its inheritance chain via a member called `"__proto__"`, the simple act of typing `"__proto__"` would cause Google Docs to lock up due to a misassignment to this member.[8] In shared documents, this enables a denial-of-service attack. The patch of concatenating the lookup string with an unambiguous prefix or suffix itself requires careful reasoning (for instance, `"_"` would be a bad choice). Broadly, such examples are a case of a *member should not be found* error. Our type system protects against these, too.

The rest of this paper presents a semantics that allows these patterns and their associated problems, types that describe these objects and their uses, and a type system that explores verifying their safe use statically.

3. A Scripting Language Object Calculus

The preceding section illustrates how objects with first-class member names are used in several scripting languages. In this section, we distill the essentials of first-class member names into an object calculus called λ_S^{ob}. λ_S^{ob} has an entirely conventional account of higher-order functions and state (figure 4). However, it has an unusual object system that faithfully models the characteristic features of objects in scripting langauges (figure 3). We also note some of the key differences between λ_S^{ob} and some popular scripting languages below.

- In member lookup $e_1[e_2]$, the member name e_2 is not a static string but an arbitrary expression that evaluates to a string. A programmer can thus dynamically pick the member name, as demonstrated in section 2.2. While it is possible to do this in languages like Java, it requires the use of cumbersome reflection APIs. Object calculi for Java thus do not bother modeling reflection. In contrast, scripting languages make dynamic member lookup easy.
- The expression $e_1[e_2 = e_3]$ has two meanings. If the member e_2 does not exist, it creates a new member (E-Create). If e_2 does exist, it updates its value. Therefore, members need not be declared and different instances of the same class or prototype can have different sets of members.
- When a member is not found, λ_S^{ob} looks for the member in the parent object, which is the subscripted v_p part of the object value (E-Inherit). In Ruby and Lua this member is not directly accessible. In JavaScript and Python, it is an actual member of the object (`"__proto__"` in JavaScript, `"__class__"` in Python).
- The o **hasfield** str expression checks if the object o has a member str anywhere on its inheritance chain. This is a basic form of reflection.
- The str **matches** P expression returns **true** if the string matches the pattern P, and **false** otherwise. Scripting language programs employ a plethora of operators to pattern match and decompose strings, as shown in section 2.2. We abstract these to a single string-matching operator and representation of string patterns. The representation of patterns P is irrelevent to our core calculus. We only require that P represent some class of string-sets with decidable membership.

Given the lack of syntactic restrictions on object lookup, we can easily write a program that looks up a member that is not defined anywhere on the inheritance chain. In such cases, λ_S^{ob} signals an error (E-NotFound). Naturally, our type system will follow the classical goal of avoiding such errors.

Soundness We mechanize λ_S^{ob} with the Coq Proof Assistant, and prove a simple untyped progress theorem.

THEOREM 1 (Progress). *If σe is a closed, well-formed configuration, then either:*

- $e \in v$,
- $e = E\langle \mathsf{err} \rangle$, *or*
- $\sigma e \to \sigma' e'$, *where $\sigma' e'$ is a closed, well-formed configuration.*

This property requires additional evaluation rules for runtime errors, which we elide from the paper.

4. Types for Objects in Scripting Languages

The rest of this paper explores typing the object calculus presented in section 3. This section addresses the structure and meaning of object types, followed by the associated type system and details of subtyping.

Typed λ_S^{ob}has explicit type annotations on variables bound by functions:

func$(x{:}T)$ { e }

Type inference is beyond the scope of this paper, so λ_S^{ob} is explicitly typed. Figure 5 specifies the full core type language. We incrementally present its significant elements, object types and string types, in the following subsections. The type language is otherwise conventional: types include base types, function types, and types for references; these are necessary to type ordinary imperative programs. We employ a top type, equirecursive μ-types, and bounded universal types to type-check object-oriented programs.

4.1 Basic Object Types

We adopt a structural type system, and begin our presentation with canonical record types, which map strings to types:

$$T = \cdots \mid \{str_1 : T_1 \cdots str_n : T_n\}$$

Record types are conventional and can type simple λ_S^{ob} programs:

```
let objWithToStr = ref {
  toStr: func(this:μα.Ref {"toStr": α → Str}) { "hello" } }
in (deref objWithToStr)["toStr"](objWithToStr)
```

We need to work a little harder to express types for the programs in section 2. We do so in a principled way—all of our additions are conservative extensions to classical record types.

[7] Arguably, the real flaw is in using the same data structure to associate both a fixed, statically-known collection of names and a dynamic, unbounded collection. Many scripting languages, however, provide only one data structure for both purposes, forcing this identification on programmers. We refrain here from moral judgment.

[8] `http://www.google.com/support/forum/p/Google+Docs/thread?tid=0cd4a00bd4aef9e4`

String patterns	P	$=$	$\cdots$	patterns are abstract, but must have decidable membership, $str \in P$
Constants	c	$=$	$bool \mid str \mid \textsf{null}$	
Values	v	$=$	c	
		$\mid$	$\texttt{\{ } str_1 \texttt{ : } v_1 \cdots str_n \texttt{ : } v_n \texttt{\}}_v$	object, where $str_1 \cdots str_n$ must be unique
Expressions	e	$=$	v	
		$\mid$	$\texttt{\{ } str_1 \texttt{ : } e_1 \cdots str_n \texttt{ : } e_n \texttt{ \}}_e$	object expression
		$\mid$	$e\texttt{[}e\texttt{]}$	lookup member in object, or in prototype
		$\mid$	$e\texttt{[}e \texttt{ = } e\texttt{]}$	update member, or create new member if needed
		$\mid$	e **hasfield** e	test if member is present
		$\mid$	e **matches** P	match a string against a pattern
		$\mid$	**err**	runtime error

$\boxed{e \hookrightarrow e}$

E-GetField	$\texttt{\{ } \cdots str\texttt{: } v \cdots \texttt{ \}}_{v_p}\texttt{[}str\texttt{]} \hookrightarrow v$
E-Inherit	$\texttt{\{ } str \texttt{ : } v \cdots \texttt{ \}}_{l_p}\texttt{[}str_x\texttt{]} \hookrightarrow \texttt{(}\textsf{deref}\ l_p\texttt{)[}str_x\texttt{]}$, if $str_x \notin (str \cdots)$
E-NotFound	$\texttt{\{ } str \texttt{ : } v \cdots \texttt{ \}}_{\textsf{null}}\texttt{[}str_x\texttt{]} \hookrightarrow \textsf{err}$, if $str_x \notin (str \cdots)$
E-Update	$\texttt{\{ } str_1\texttt{: } v_1 \cdots\ str_i\texttt{: } v_i\ \cdots str_n\texttt{: } v_n\ \texttt{\}}_{v_p}\texttt{[}str_i \texttt{ = } v\texttt{]} \hookrightarrow \texttt{\{ } str_1\texttt{: } v_1 \cdots\ str_i\texttt{: } v\ \cdots str_n\texttt{: } v_n\ \texttt{\}}_{v_p}$
E-Create	$\texttt{\{ } str_1\texttt{: } v_1 \cdots \texttt{\}}_{v_p}\texttt{[}str_x \texttt{ = } v_x\texttt{]} \hookrightarrow \texttt{\{ } str_x\texttt{: } v_x\texttt{, } str_1\texttt{: } v_1 \cdots \texttt{\}}_{v_p}$ when $str_x \notin (str_1 \cdots)$
E-Hasfield	$\texttt{\{}\cdots str\texttt{:}v \cdots \texttt{ \}}_{v_p}$ **hasfield** $str \hookrightarrow$ **true**
E-HasFieldProto	$\texttt{\{}\cdots str\texttt{:}v \cdots \texttt{ \}}_{v_p}$ **hasfield** $str \hookrightarrow v_p$ **hasfield** str, when $str' \notin (str \cdots)$
E-HasNotField	**null hasfield** $str \hookrightarrow$ **false**
E-Matches	str **matches** $P \hookrightarrow$ **true**, $str \in P$
E-NoMatch	str **matches** $P \hookrightarrow$ **false**, $str \notin P$

Figure 3. Semantics of Objects and String Patterns in λ_S^{ob}

Locations	l	$=$	$\cdots$	heap addresses
Heaps	σ	$=$	$\cdot \mid (l, v)\sigma$	
Values	v	$=$	$\cdots \mid l \mid \textsf{func}\texttt{(}x\texttt{) \{ } e \texttt{ \}}$	
Expressions	e	$=$	$\cdots$	
		$\mid$	x	identifiers
		$\mid$	$e\texttt{(}e\texttt{)}$	function application
		$\mid$	$e \texttt{ := } e$	update heap
		$\mid$	**ref** e	initialize a new heap location
		$\mid$	**deref** e	heap lookup
		$\mid$	**if** $\texttt{(}e_1\texttt{)}\ e_2$ **else** e_3	branching
Evaluation Contexts	E	$=$	$\cdots$	left-to-right, call-by-value evaluation

$\boxed{e \hookrightarrow e}$

β_v	$\texttt{(}\textsf{func}\texttt{(}x\texttt{) \{ } e \texttt{ \})(}v\texttt{)} \hookrightarrow e[x/v]$
E-IfTrue	**if** $\texttt{(}\textsf{true}\texttt{)}\ e_2$ **else** $e_3 \hookrightarrow e_2$
E-IfFalse	**if** $\texttt{(}\textsf{false}\texttt{)}\ e_2$ **else** $e_3 \hookrightarrow e_3$

$\boxed{\sigma e \rightarrow \sigma e}$

E-Cxt	$\sigma E\langle e_1 \rangle \rightarrow \sigma E\langle e_2 \rangle$, when $e_1 \hookrightarrow e_2$
E-Ref	$\sigma E\langle \textsf{ref}\ v \rangle \rightarrow \sigma(l, v)E\langle l \rangle$, when $l \notin dom(\sigma)$
E-Deref	$\sigma E\langle \textsf{deref}\ l \rangle \rightarrow \sigma E\langle \sigma(l) \rangle$
E-SetRef	$\sigma E\langle l \texttt{ := } v \rangle \rightarrow \sigma[l := v]E\langle v \rangle$, when $l \in dom(\sigma)$

Figure 4. Conventional Features of λ_S^{ob}

4.2 Dynamic Member Lookups: String Patterns

The previous example is uninteresting because all member names are statically specified. Consider a Beans-inspired example, where there are potentially many members defined as `"get.*"` and `"set.*"`. Beans libraries construct actual calls to methods by getting property names from reflection or configuration files, and appending them to the strings `"get"` and `"set"` before invoking. Beans also inherit useful built-in functions, like `"toString"` and `"hashCode"`.

We need some way to represent all the potential get and set members on the object. Rather than a collection of singleton names, we need families of member names. To tackle this, we begin by introducing string patterns, rather than first-order labels, into our object types:

String patterns	L	$=$	P
String and object types	T	$=$	$\cdots \mid L \mid \{L_1 : T_1 \cdots L_n : T_n\}$

Patterns, P, represent sets of strings. String literals type to singleton string sets, and subsumption is defined by set inclusion.

String patterns	L	$=$	P	extended in figure 11
Base types	b	$=$	$\mathsf{Bool} \mid \mathsf{Null}$	
Type variables	α	$=$	$\cdots$	
Types	T	$=$	b	
		$\mid$	$T_1 \to T_2$	function types
		$\mid$	$\mu\alpha.T$	recursive types
		$\mid$	$\mathsf{Ref}\ T$	type of heap locations
		$\mid$	$\top$	top type
		$\mid$	L	string types
		$\mid$	$\{L_1^{p_1} : T_1, \cdots, L_n^{p_n} : T_n, L_A : \mathbf{abs}\}$	object types
Member presence	p	$=$	$\downarrow$	definitely present
		$\mid$	$\circ$	possibly absent
Type Environments	Γ	$=$	$\cdot \mid \Gamma, x : T$	

$\boxed{\Gamma \vdash T}$

$$\text{WF-Object}\ \frac{\Gamma \vdash T_1 \cdots \Gamma \vdash T_n \quad \forall i. L_i \cap L_A = \emptyset \text{ and } \forall j \neq i. L_i \cap L_j = \emptyset}{\Gamma \vdash \{L_1^{p_1} : T_1 \cdots L_n^{p_n} : T_n, L_A : \mathbf{abs}\}}$$

The well-formedness relation for types is mostly conventional. We require that string patterns in an object must not overlap (WF-Object).

Figure 5. Types for λ_S^{ob}

Our theory is parametric over the representation of patterns, as long as pattern containment, $P_1 \subseteq P_2$ is decidable, and the pattern language is closed over the following operators:

$$P_1 \cap P_2 \qquad P_1 \cup P_2 \qquad \overline{P} \qquad P_1 \subseteq P_2 \qquad P = \emptyset$$

With this notation, we can write an expressive type for our Bean-like objects, assuming Int-typed getters and setters:[9]

$$\mathsf{IntBean} = \{ \quad (\texttt{"get".*}) :\to \mathsf{Int}, \\ (\texttt{"set".*}) : \mathsf{Int} \to \mathsf{Null}, \texttt{"toString"} :\to \mathsf{Str} \quad \}$$

(where `.*` is the regular expression pattern of all strings, so `"get".*` is the set of all strings prefixed by get). Note, however, that IntBean seems to promise the presence of an infinite number of members, which no real object has. This type must therefore be interpreted to mean that a getter, say, will have Int *if it is present*. Since it may be absent, we can get the very "member not found" error that the type system was trying to *prevent*, resulting in a failure to conservatively extend simple record types.

In order to model members that we know are safe to look up, we add annotations to members that indicate whether they are *definitely* or *possibly* present:

Member presence	p	$=$	$\downarrow \mid \circ$
Object types	T	$=$	$\cdots \mid \{L_1^{p_1} : T_1 \cdots L_n^{p_n} : T_n\}$

The notation $L^{\downarrow} : T$ means that for each string $\mathsf{str} \in L$, a member with name str *must* be present on the object, and the value of the member has type T. This is the traditional meaning of a member's annotation in simple record types. In contrast, $L^{\circ} : T$ means that for each string $\mathsf{str} \in L$, *if* there is a member named str on the object, *then* the member's value has type T. We would therefore write the above type as:

$$\mathsf{IntBean} = \{ \quad (\texttt{"get".*})^{\circ} :\to \mathsf{Int}, (\texttt{"set".*})^{\circ} : \mathsf{Int} \to \mathsf{Null}, \\ \texttt{"toString"}^{\downarrow} :\to \mathsf{Str} \quad \}$$

As a matter of well-formedness, it does not make sense to place a definitely-present annotation ($\downarrow$) on an infinite set of strings, only on finite ones (such as the singleton set consisting of `"toString"` above). In contrast, possibly-present annotations can be placed even on finite sets: writing $\texttt{"toString"}^{\circ}$ would indicate that the toString member does not need to be present.

Definitely-present members allow us to recover simple record types. However, we still cannot guarantee safe lookup within an infinite set of member names. We will return to this problem in section 4.6.

4.3 Subtyping

Once we introduce record types with string pattern names, it is natural to ask how pairs of types relate. This relationship is important in determining, for instance, when an actual argument may safely be passed to a formal parameter. This requires the definition of a subtyping relationship.

There are two well-known kinds of subsumption rules for record types, popularly called *width* and *depth* subtyping [1]. Depth subtyping allows for specialization, while width subtyping hides information. Both are useful in our setting, and we would like to understand them in the context of first-class member names. String patterns and possibly-present members introduce more cases to consider, and we must account for them all.

When determining whether one object type can be subsumed by another, we must consider whether each member can be subsumed. We will therefore treat each member name individually, iterating over all strings. Later, we will see how we can avoid iterating over this infinite set.

Figure 6 presents the initial version of our subtyping relation. For each member s, it considers the member's annotation in each of the subtype and supertype. Each member name can have one of three relationships with a type T: definitely present in T, possibly present in T, or not mentioned at all (indicated by a dash, $-$). This naturally results in nine combinations to consider.

The column labeled *Antecedent* describes what further proof is needed to show subsumption. We use *ok* to indicate axiomatic base cases of the subtyping relation, ☠ to indicate cases where subsumption is undefined (resulting in a "subtyping" error), and $S <: T$ to indicate what is needed to show that the two members subsume. In the explanation below, we refer to table rows by the annotations on the members: e.g., $s^{\downarrow}s^{\downarrow}$ refers to the first row and $s^{\circ}s-$ to the sixth.

[9] Adding type abstraction at the member level gives more general setters and getters, but is orthogonal to our goals in this example.

Subtype	*Supertype*	*Antecedent*
$s^{\downarrow} : S$	$s^{\downarrow} : T$	$S <: T$
$s^{\downarrow} : S$	$s^{\circ} : T$	$S <: T$
$s^{\downarrow} : S$	$s : -$	ok
$s^{\circ} : S$	$s^{\downarrow} : T$	💀
$s^{\circ} : S$	$s^{\circ} : T$	$S <: T$
$s^{\circ} : S$	$s : -$	ok
$s : -$	$s^{\downarrow} : T_p$	💀
$s : -$	$s^{\circ} : T_p$	💀
$s : -$	$s : -$	ok

Figure 6. Per-String Subsumption (incomplete, see figure 7)

Depth Subtyping All the cases where further proof that $S <: T$ is needed are instances of depth subtyping: $s^{\downarrow}s^{\downarrow}$, $s^{\downarrow}s^{\circ}$, and $s^{\circ}s^{\circ}$. The $s^{\circ}s^{\downarrow}$ case is an error because a possibly-present member cannot automatically become definitely-present: e.g., substituting a value of type $\{\texttt{"x"}^{\circ} : \mathsf{Int}\}$ where $\{\texttt{"x"}^{\downarrow} : \mathsf{Int}\}$ is expected is unsound because the member may fail to exist at run-time. However, a member's presence can gain in strength through an operation that checks for the existence of the named member; we return to this point in section 4.6.

Width subtyping In the first two ok cases, $s^{\downarrow}s\text{-}$ and $s^{\circ}s\text{-}$, a member is "dropped" by being left unspecified in the supertype. This corresponds to information hiding.

The Other Three Cases We have discussed six of the cases above. Of the remaining three, $s\text{-}s\text{-}$ is the simple reflexive case. The other two, $s\text{-}s^{\downarrow}$ and $s\text{-}s^{\circ}$, must be errors because the subtype has failed to name a member (which may in fact be present), thereby attempting information hiding; if the supertype reveals this member, it would leak that information.

4.3.1 A Subtyping Parable

For a moment, let us consider the classical object type world with fixed sets of members and no presence annotations [1]. We will use patterns purely as a syntactic convenience, i.e., they will represent only a finite set of strings (which we could have written out by hand). Consider the following neutered array of booleans, which has only ten legal indices:

$$\mathsf{DigitArray} \equiv \{(\texttt{[0-9]}) : \mathsf{Bool}\}$$

Clearly, this type can be subsumed to an even smaller one of just three indices:

$$\{(\texttt{[0-9]}) : \mathsf{Bool}\} <: \{(\texttt{[1-3]}) : \mathsf{Bool}\}$$

Now consider the following object, with a proposed type that would be ascribed by a classical type rule for object literals:

$$\begin{array}{rl} \mathsf{obj} = & \{\texttt{"0"}\texttt{:}\ \textbf{false}\texttt{,}\ \texttt{"1"}\texttt{:}\ \textbf{true}\}_{\textbf{null}} \\ \mathsf{obj} : & \{\texttt{[0-1]} : \mathsf{Bool}\} \end{array}$$

obj clearly does not have type DigitArray, since it lacks eight required members. Suppose, using the more liberal type system of this paper, which permits possibly-present members, we define the following more permissive array:

$$\mathsf{DigitArrayMaybe} \equiv \{(\texttt{[0-9]})^{\circ} : \mathsf{Bool}\}$$

	Subtype	*Supertype*	*Antecedent*
	$s^{\downarrow} : S$	$s^{\downarrow} : T$	$S <: T$
	$s^{\downarrow} : S$	$s^{\circ} : T$	$S <: T$
	$s^{\downarrow} : S$	$s : -$	ok
→	$s^{\downarrow} : S$	$s : \mathbf{abs}$	💀
	$s^{\circ} : S$	$s^{\downarrow} : T$	💀
	$s^{\circ} : S$	$s^{\circ} : T$	$S <: T$
	$s^{\circ} : S$	$s : -$	ok
→	$s^{\circ} : S$	$s : \mathbf{abs}$	💀
	$s : -$	$s^{\downarrow} : T_p$	💀
	$s : -$	$s^{\circ} : T_p$	💀
	$s : -$	$s : -$	ok
→	$s : -$	$s : \mathbf{abs}$	💀
→	$s : \mathbf{abs}$	$s^{\downarrow} : T$	💀
→	$s : \mathbf{abs}$	$s^{\circ} : T$	ok
→	$s : \mathbf{abs}$	$s : -$	ok
→	$s : \mathbf{abs}$	$s : \mathbf{abs}$	ok

Figure 7. Per-String Subsumption (with Absent Fields)

By the $s\text{-}s^{\circ}$ case, obj's type doesn't subsume to DigitArrayMaybe, either, with good reason: something of its type could be hiding the member "2" containing a string, which if dereferenced (as DigitArrayMaybe permits) would result in a type error at run-time. (Of course, this does not tarnish the utility of maybe-present annotations, which we introduced to handle infinite sets of member names.)

However, there is information about obj that we have not captured in its type: namely that the members not listed *truly are absent*. Thus, though obj still cannot satisfy DigitArray (or, equivalently in our notation, $\{(\texttt{[0-9]})^{\downarrow} : \mathsf{Bool}\}$), it is reasonable to permit the value obj to inhabit the type DigitArrayMaybe, whose client understands that some members may fail to be present at run-time. We now extend our type language to permit this flexibility.

4.3.2 Absent Fields

To model absent members, we augment our object types one step further to describe the set of member names that are *definitely absent* on an object:

$$\begin{array}{rcl} p & = & \downarrow \mid \circ \\ T & = & \cdots \mid \{L_1^{p_1} : T_1 \cdots L_n^{p_n} : T_n, L_A : \mathbf{abs}\} \end{array}$$

With this addition, there is now a fourth kind of relationship a string can have with an object type: it can be known to be absent. We must update our specification of subtyping accordingly. Figure 7 has the complete specification, where the new rows are marked with an arrow →.

Nothing subsumes *to* $\mathbf{abs}$ except for $\mathbf{abs}$ itself, so supertypes have a (non-strict) subset of the absent members of their subtypes. This is enforced by cases $s^{\downarrow}s^{\mathbf{a}}$, $s^{\circ}s^{\mathbf{a}}$, and $s\text{-}s^{\mathbf{a}}$ (where we use $s^{\mathbf{a}}$ as the abbreviation for an $s : \mathbf{abs}$ entry).

If a string is absent on the subtype, the supertype cannot claim it is definitely present ($s^{\mathbf{a}}s^{\downarrow}$). In the last three cases ($s^{\mathbf{a}}s^{\circ}$, $s^{\mathbf{a}}s\text{-}$, or $s^{\mathbf{a}}s^{\mathbf{a}}$), an absent member in the subtype can be absent, not mentioned, or possibly-present with *any* type in the supertype,

Child	*Parent*	*Result*
$s^{\downarrow} : T_c$	$s^{\downarrow} : T_p$	$s^{\downarrow} : T_c$
$s^{\downarrow} : T_c$	$s^{\circ} : T_p$	$s^{\downarrow} : T_c$
$s^{\downarrow} : T_c$	$s : -$	$s^{\downarrow} : T_c$
$s^{\downarrow} : T_c$	$s : \mathbf{abs}$	$s^{\downarrow} : T_c$
$s^{\circ} : T_c$	$s^{\downarrow} : T_p$	$s^{\downarrow} : T_c \sqcup T_p$
$s^{\circ} : T_c$	$s^{\circ} : T_p$	$s^{\circ} : T_c \sqcup T_p$
$s^{\circ} : T_c$	$s : -$	$s : -$
$s^{\circ} : T_c$	$s : \mathbf{abs}$	$s^{\circ} : T_c$
$s : -$	$s^{\downarrow} : T_p$	$s : -$
$s : -$	$s^{\circ} : T_p$	$s : -$
$s : -$	$s : -$	$s : -$
$s : -$	$s : \mathbf{abs}$	$s : -$
$s : \mathbf{abs}$	$s^{\downarrow} : T_p$	$s^{\downarrow} : T_p$
$s : \mathbf{abs}$	$s^{\circ} : T_p$	$s^{\circ} : T_p$
$s : \mathbf{abs}$	$s : \mathbf{abs}$	$s : \mathbf{abs}$
$s : \mathbf{abs}$	$s : -$	$s : -$

Figure 8. Per-String Flattening

which even allows subsumption to introduce types for members that may be added in the future. To illustrate our final notion of subsumption, we use a more complex version of the earlier example (overline indicates set complement):

$$\begin{aligned} \mathsf{BoolArray} \equiv\ & \{(\texttt{[0-9]}^{+})^{\circ} : \mathsf{Bool}, \texttt{"length"}^{\downarrow} : \mathsf{Num}\} \\ \mathsf{arr} \equiv\ & \texttt{\{"0": false, "1": true, "length":2\}}_{\mathsf{null}} \end{aligned}$$

Suppose we ascribe the following type to arr:

$$\mathsf{arr} : \quad \{ \quad \texttt{[0-1]}^{\downarrow} : \mathsf{Bool}, \texttt{"length"}^{\downarrow} : \mathsf{Num}, \quad \overline{\{\texttt{[0-1]}, \texttt{"length"}\}} : \mathbf{abs} \quad \}$$

This subsumes to BoolArray thus:

- The member `"length"` is subsumed using $s^{\downarrow}s^{\downarrow}$, with $\mathsf{Num} <: \mathsf{Num}$.
- The members `"0"` and `"1"` subsume using $s^{\downarrow}s^{\circ}$, with $\mathsf{Bool} <: \mathsf{Bool}$.
- The members made of digits other than `"0"` and `"1"` (as a regex, `[0-9][0-9]`$^{+} \cup$ `[2-9]`), subsume using $s^{\mathbf{a}}s^{\circ}$, where T is Bool.
- The remaning members—those that aren't `"length"` and whose names aren't strings of digits—such as `"iwishiwereml"` are hidden by $s^{\mathbf{a}}s$-.

Absent members let us bootstrap from simple object types into the domain of infinite-sized collections of possibly-present members. Further, we recover simple record types when $L_A = \emptyset$.

4.3.3 Algorithmic Subtyping

Figure 7 gives a declarative specification of the per-string subtyping rules. This is clearly not an algorithm, since it iterates over pairs of an infinite number of strings. In contrast, our object types contain string patterns, which are finite representations of infinite sets. By considering the pairwise intersections of *patterns* between the two object types, an algorithmic approach presents itself. The algorithmic typing judgments must contain clauses that work at the level of

Child	*Parent*	*Antecedent*
$s^{\downarrow} : T_c$	$s^{\downarrow} : T_p$	$T_c <: T_p$
$s^{\downarrow} : T_c$	$s^{\circ} : T_p$	$T_c <: T_p$
$s^{\downarrow} : T_c$	$s : -$	ok
$s^{\downarrow} : T_c$	$s : \mathbf{abs}$	☠
$s^{\circ} : T_c$	$s^{\downarrow} : T_p$	$T_c \sqcup T_p <: T_p$ $(= T_c <: T_p)$
$s^{\circ} : T_c$	$s^{\circ} : T_p$	$T_c \sqcup T_p <: T_p$ $(= T_c <: T_p)$
$s^{\circ} : T_c$	$s : -$	ok
$s^{\circ} : T_c$	$s : \mathbf{abs}$	☠
$s : -$	$s^{\downarrow} : T_p$	☠
$s : -$	$s^{\circ} : T_p$	☠
$s : -$	$s : -$	ok
$s : -$	$s : \mathbf{abs}$	☠
$s : \mathbf{abs}$	$s^{\downarrow} : T_p$	$T_p <: T_p$ $(= ok)$
$s : \mathbf{abs}$	$s^{\circ} : T_p$	$T_p <: T_p$ $(= ok)$
$s : \mathbf{abs}$	$s : -$	ok
$s : \mathbf{abs}$	$s : \mathbf{abs}$	ok

Figure 9. Subtyping after Flattening

patterns. Thus, in deciding subtyping for

$$\{L_1^{p_1} : S_1, \cdots, L_A : \mathbf{abs}\} <: \{M_1^{q_1} : T_1, \cdots, M_A : \mathbf{abs}\}$$

one of several antecedents intersects all the definitely-present patterns, and checks that they are subtypes:

$$\forall i, j.(L_i \cap M_j \neq \emptyset) \wedge (p_i = q_j = \downarrow) \implies S_i <: T_i$$

A series of rules like these specify an algorithm for subtyping that is naturally derived from figure 7. The full definition of algorithmic subtyping is available online.

4.4 Inheritance

Conventional structural object types do not expose the position of members on the inheritance chain; types are "flattened" to include inherited members. A member lower in the chain "shadows" one of the same name higher in the chain, with only the lower member's type present in the resulting record.

The same principle applies to first-class member names but, as with subtyping, we must be careful to account for all the cases. For subtyping, we related subtypes and supertypes to the proof obligation needed for their subsumption. For flattening, we will define a function that constructs a new object type out of two existing ones:

$$\mathit{flatten} \in T \times T \to T$$

As with subtyping, it suffices to specify what should happen for a given member s. Figure 8 shows this specification.

- When the child has a *definitely present* member, it overrides the parent ($s^{\downarrow}s^{\downarrow}$, $s^{\downarrow}s^{\circ}$, $s^{\downarrow}s$-, and $s^{\downarrow}s^{\mathbf{a}}$).

- In the cases $s^\circ s^\downarrow$ and $s^\circ s^\circ$, the child specifies a member as *possibly present* and the parent also specifies a type for the member, so a lookup may produce a value of either type. Therefore, we must join the two types (the $\sqcup$ operator).[10] In case $s^\circ s\text{-}$, the parent doesn't specify the member; it cannot be safely overridden with only a possibly-present member, so we must leave it hidden in the result. In case $s^\circ s^{\mathbf{a}}$, since the member is absent on the parent, it is left possibly-present in the result.
- If the child doesn't specify a member, it hides the corresponding member in the parent ($s\text{-}s^\downarrow$, $s\text{-}s^\circ$, $s\text{-}s\text{-}$, $s\text{-}s^{\mathbf{a}}$).
- If a member is absent on the child, the corresponding member on the parent is used ($s^{\mathbf{a}}s^\downarrow$, $s^{\mathbf{a}}s^\circ$, $s^{\mathbf{a}}s\text{-}$, $s^{\mathbf{a}}s^{\mathbf{a}}$).

The online material includes a flattening algorithm.

Inheritance and Subtyping It is well-known that inheritance and subtyping are different, yet not completely orthogonal concepts [9]. Figures 7 and 8 help us identify when an object inheriting from a parent is a subtype of that parent. Figure 9 presents this with three columns. The first two show the presence and type in the child and parent, respectively. In the third column, we apply *flatten* to that row's child and parent; then look up the result and the parent's type in the figure 7; and copy the resulting *Antecedent* entry. This column thus explains under exactly what condition a child that extends a parent is a subtype of it. Consider some examples:

- If a child overrides a parent's member (e.g., $s^\downarrow s^\downarrow$), it must override the member with a subtype.
- When the child hides a parent's definitely-present member ($s\text{-}s^\downarrow$), it is not a subtype of its parent.
- Suppose a parent has a definitely-present member s, which is explicitly not present in the child. This corresponds to the $s^{\mathbf{a}}s^\downarrow$ entry. Applying *flatten* to these results in $s^\downarrow : T_p$. Looking up and substituting $s^\downarrow : T_p$ for both subtype and supertype in figure 7 yields the condition $T_p <: T_p$, which is always true. Indeed, at run-time the parent's s would be accessible through the child, and (for this member) inheritance would indeed correspond to subtyping.

By relating pairs of types, this table could, for instance, determine whether a mixin—which would be represented here as an object-to-object function that constructs objects relative to a parameterized parent—obeys subtyping.

4.5 Typing Objects

Now that we are equipped with *flatten* and a notion of subsumption, we can address typing λ_S^{ob} in full. Figure 10 contains the rules for typing strings, objects, member lookup, and member update.

T-Str is straightforward: literal strings have a singleton string set L-type. T-Object is more interesting. In a literal object expression, all the members in the expression are definitely present, and have the type of the corresponding member expression; these are the pairs $str_i^\downarrow : S_i$. Fields not listed are definitely absent, represented by $\star : \mathbf{abs}$.[11] In addition, object expressions have an explicit parent subexpression (e_p). Using *flatten*, the type of the new object is combined with that of the parent, T_p.

A member lookup expression has the type of the member corresponding to the lookup member pattern. T-GetField ensures that the type of the expression in lookup position is a set of definitely-present members on the object, L_i, and yields the corresponding type, S_i.

The general rule for member update, T-Update, similarly requires that the entire pattern L_i be on the object, and ensures invariance of the object type under update (T is the type of e_o and the type of the whole expression). In contrast to T-GetField, the member does *not* need to be definitely-present: assigning into possibly-present members is allowed, and maintains the object's type.

Since simple record types allow strong update by extension, our type system admits one additional form of update expression, T-UpdateStr. If the pattern in lookup position has a singleton string type str_f, then the resulting type has str_f definitely present with the type of the update argument, and removes str_f from each existing pattern.

4.6 Typing Membership Tests

Thus far, it is a type error to lookup a member that is possibly absent all along the inheritance chain: T-GetField requires that the accessed member is definitely present. For example, if `obj` is a dictionary mapping member names to numbers:

$$\{(.*)^\circ : \mathsf{Num}, \emptyset : \mathbf{abs}\}$$

then `obj["10"]` is untypable. We can of course relax this restriction and admit a runtime error, but it would be better to give a guarantee that such an error cannot occur. A programmer might implement a lookup wrapper with a guard:

if (obj **hasfield** x) obj[x] **else false**

Our type system must account for such guards and *if-split*—it must narrow the types of `obj` and `x` appropriately in at least the then-branch. We present an single if-splitting rule that exactly matches this pattern:[12]

if $(x_{obj}$ **hasfield** $x_{fld})$ e_2 **else** e_3

A special case is when the type of x_{obj} is $\{\cdots L^\circ : T \cdots\}$ and the type of x_{fld} is a singleton string $str \in L$. In such a case, we can narrow the type of x_{obj} to:

$$\{\cdots \{str\}^\downarrow : T, L \cap \overline{\{str\}}^\circ : T \cdots\}$$

A lookup on this type with the string *str* would then be typable with type T. However, the second argument to **hasfield** won't always type to a singleton string. In this case, we need to use a bounded type variable to represent it. We enrich string types to represent this, shown in the if-splitting rule in figure 11.[13] For example, let $P = (.*)$. If $\mathsf{x} : P$ and $\mathsf{obj} : \{P^\circ : \mathsf{Num}, \emptyset : \mathbf{abs}\}$, then the narrowed environment in the true branch is:

$$\Gamma' = \Gamma, \alpha <: P, \mathsf{x} : \alpha, \mathsf{obj} : \{\alpha^\downarrow : \mathsf{Num}, P \cap \overline{\alpha}^\circ : \mathsf{Num}, \emptyset : \mathbf{abs}\}$$

This new environment says that `x` is bound to a value that types to some subset of P, and that subset is definitely present ($\alpha^\downarrow$) on the object `obj`. Thus, a lookup `obj[x]` is guaranteed to succeed with type Num.

Object subtyping must now work with the extended string patterns of figure 11, introducing a dependency on the environment Γ. Instead of statements such as $L_1 \subseteq L_2$, we must now discharge $\Gamma \vdash L_1 \subseteq L_2$. We interpret these as propositions with set variables that can be discharged by existing string solvers [19].

[10] A type U is the join of S and T if $S <: U$ and $T <: U$.

[11] In our type language, we use $\star$ to represent "all members not named in the rest of the object type". Since string patterns are closed over union and negation, $\star$ can be expressed directly as the complement of the union of the other patterns in the object type. Therefore $\star$ is purely a syntactic convenience, and does not need to appear in the theory.

[12] There are various if-splitting techniques for typing complex conditionals and control [8, 16, 34]. We regard the details of these techniques as orthogonal.

[13] This single typing rule is adequate for type-checking, but the proof of preservation requires auxiliary rules in the style of Typed Scheme [33].

$$\text{T-Str}\ \frac{}{\Sigma;\Gamma \vdash str : \{str\}} \qquad \text{T-Object}\ \frac{\Sigma;\Gamma \vdash e_1 : S_1 \cdots \qquad \Sigma;\Gamma \vdash e_p : \mathsf{Ref}\ T_p \qquad T = \mathit{flatten}(\{str_1^{\downarrow} : S_1 \cdots \star : \mathbf{abs}\}, T_p)}{\Sigma;\Gamma \vdash \texttt{\{}\ str_1\texttt{:}\ e_1 \cdots\ \texttt{\}}_{e_p} : T}$$

$$\text{T-GetField}\ \frac{\Sigma;\Gamma \vdash e_o : \{L_1^{p_1} : S_1, \cdots, L_i^{\downarrow} : S_i, \cdots, L_n^{p_n} : S_n, L_A : \mathbf{abs}\} \qquad \Sigma;\Gamma \vdash e_f : L_i}{\Sigma;\Gamma \vdash e_o\texttt{[}e_f\texttt{]} : S_i}$$

$$\text{T-Update}\ \frac{\Sigma;\Gamma \vdash e_o : T \qquad \Sigma;\Gamma \vdash e_f : L_i \qquad \Sigma;\Gamma \vdash e_v : S_i \qquad T = \{L_1^{p_1} : S_1, \cdots, L_i^{p_i} : S_i, \cdots L_n^{p_n} : S_n, L_A : \mathbf{abs}\}}{\Sigma;\Gamma \vdash e_o\texttt{[}e_f\ \texttt{=}\ e_v\texttt{]} : T}$$

$$\text{T-UpdateStr}\ \frac{\Sigma;\Gamma \vdash e_f : \{str_f\} \qquad \Sigma;\Gamma \vdash e_v : S \qquad \Sigma;\Gamma \vdash e_o : \{L_1^{p_1} : S_1, \cdots, L_A : \mathbf{abs}\}}{\Sigma;\Gamma \vdash e_o\texttt{[}e_f\ \texttt{=}\ e_v\texttt{]} : \{str_f^{\downarrow} : S, (L_1 - str_f)^{p_1} : S_1, \cdots, L_A - str_f : \mathbf{abs}\}}$$

Figure 10. Typing Objects

$$\begin{array}{lrcll} \text{String patterns} & L & = & P \mid \alpha \mid L \cap L \mid L \cup L \mid \overline{L} & \\ \text{Types} & T & = & \cdots \mid \forall \alpha <: S.T & \text{bounded quantification} \\ \text{Type Environments} & \Gamma & = & \cdots \mid \Gamma, \alpha <: T & \end{array}$$

$$\frac{\Gamma(o) = \{\cdots L^{\circ} : S \cdots\} \qquad \Gamma(f) = L \qquad \Sigma;\Gamma, \alpha <: L, f : \alpha, o : \{\cdots \alpha^{\downarrow} : S, L'^{\circ} : S \cdots\} \vdash e_2 : T \qquad L' = L \cap \overline{\alpha} \qquad \Gamma \vdash e_3 : T}{\Sigma;\Gamma \vdash \textbf{if}\ (o\ \textbf{hasfield}\ f)\ e_2\ \textbf{else}\ e_3 : T}$$

Figure 11. Typing Membership Tests

4.7 Accessing the Inheritance Chain

In λ_S^{ob}, the inheritance chain of an object is present in the model but not provided explicitly to the programmer. Some scripting languages, however, expose the inheritance chain through members: `"__proto__"` in JavaScript and `"__class__"` in Python (section 2). Reflecting this in the model introduces significant complexity into the type system and semantics, which need to reason about lookups and updates that conflict with this explicit parent member. We have therefore elided this feature from the presentation in this paper. In the appendix, however, we present a version of λ_S^{ob} where a member `"parent"` is explicitly used for inheritance. All our proofs work over this richer language.

4.8 Object Types as Intersections of Dependent Refinements

An alternate scripting semantics might encode objects as functions, with application as member lookup. Thus an object of type:

$$\{L_1^{\downarrow} : T_1, L_2^{\downarrow} : T_2, \emptyset : \mathbf{abs}\}$$

could be encoded as a function of type:

$$(\{s : \mathsf{Str} | s \in L_1\} \to T_1) \wedge (\{s : \mathsf{Str} | s \in L_2\} \to T_2)$$

Standard techniques for typing intersections and dependent refinements would then apply.

This trivial encoding precludes precisely typing the inheritance chain and member presence checking in general. In contrast, the extensionality of records makes reflection much easier. However, we believe that a typable encoding of objects as functions is related to the problem of typing object proxies (which are present in Ruby and Python and will soon be part of JavaScript), which are collections of functions that behave like objects but are not objects themselves [11]. Proxies are beyond the scope of this paper, but we note their relationship to dependent types.

4.9 Guarantees

The full typing rules and formal definition of subtyping are available online. The elided typing rules are a standard account of functions, mutable references, and bounded quantification. Subtyping is interpreted co-inductively, since it includes equirecursive μ-types. We prove the following properties of λ_S^{ob}:

LEMMA 1 (Decidability of Subtyping). *If the functions and predicates on string patterns are decidable, then the subtype relation is finite-state.*

THEOREM 2 (Preservation). *If:*

- $\Sigma \vdash \sigma$,
- $\Sigma; \cdot \vdash e : T$, *and*
- $\sigma e \to \sigma' e'$,

then there exists a Σ', *such that:*

- $\Sigma \subseteq \Sigma'$,
- $\Sigma' \vdash \sigma'$, *and*
- $\Sigma'; \cdot \vdash e' : T$.

THEOREM 3 (Typed Progress). *If* $\Sigma \vdash \sigma$ *and* $\Sigma; \cdot \vdash e : T$ *then either* $e \in v$ *or there exist* σ' *and* e' *such that* $\sigma e \to \sigma' e'$.

Unlike the untyped progress theorem of section 3, this typed progress theorem does not admit runtime errors.

5. Implementation and Uses

Though our main contribution is intended to be foundational, we have built a working type-checker around these ideas, which we now discuss briefly.

Our type checker uses two representations for string patterns. In both cases, a type is represented a set of pairs, where each pair is a set of member names and their type. The difference is in the representation of member names:

```
type Array =
 typrec array :: * => * .
  typlambda a :: * . {
    /(([0-9])+|("+Infinity"|("-Infinity"|"NaN")))/ : 'a,
    length : Int,
    * : _,
    __proto__: {
      __proto__: Object,
      * : _,
      // Note how the type of "this" is array<'a>. If
      // these are applied as methods, arr.map(...), then
      // the inner 'a and outer 'a will be the same.
      map: forall a . forall b . ['array<'a>] ('a -> 'b)
           -> 'array<'b>,
      slice: forall a . ['array<'a>] Int * Int + Undef
             -> 'array<'a>,
      concat: forall a . ['array<'a>] 'array<'a>
              -> 'array<'a>,
      forEach: forall a . ['array<'a>] ('a -> Any)
               -> Undef,
      filter: forall a . ['array<'a>] ('a -> Bool)
              -> 'array<'a>,
      every: forall a . ['array<'a>] ('a -> Bool)
             -> Bool,
      some: forall a . ['array<'a>] ('a -> Bool)
            -> Bool,
      push: forall a . ['array<'a>] 'a -> Undef,
      pop: forall a . ['array<'a>] -> 'a,
      /* and several more methods */
    }
 }
```

Figure 12. JavaScript Array Type (Fragment)

1. The set of member names is a finite enumeration of strings representing either a collection of members or their complement.
2. The set of member names is represented as an automaton whose language is that set of names.

The first representation is natural when objects contain constant strings for member names. When given infinite patterns, however, our implementation parses their regular expressions and constructs automata from them.

The subtyping algorithms require us to calculate several intersections and complements. This means we might, for instance, need to compute the intersection of a finite set of strings with an automaton. In all such cases, we simply construct an automaton out of the finite set of strings and delegate the computation to the automaton representation.

For finite automata, we use the representation and decision procedure of Hooimeijer and Weimer [20]. Their implementation is fast and based on mechanically proven principles. Because our type checker internally converts between the two representations, the treatment of patterns is thus completely hidden from the user.[14]

Of course, a practical type checker must address more issues than just the core algorithms. We have therefore embedded these ideas in the existing prototype JavaScript type-checker of Guha, et al. [15, 16]. Their checker already handles various details of JavaScript source programs and control flow, including a rich treatment of if-splitting [16], which we can exploit. In contrast, their (undocumented) object types use simple record types that can only type trivial programs. Prior applications of that type-checker to most real-world JavaScript programs has depended on the theory in this paper.

We have applied this type system in several contexts:

- We can type variants of the examples in this paper; because we lack parsers for the different input languages, they are implemented in λ_S^{ob}. These examples are all available in our open source implementation as test cases.[15]
- Our type system is rich enough to provide an accurate type for complex built-in objects such as JavaScript's arrays (an excerpt from the actual code is shown in figure 12; the top of this type uses standard type system features that we don't address in this paper). Note that patterns enable us to accurately capture not only numeric indices but also JavaScript oddities such as the Infinity that arises from overflow.
- We have applied the system to type-check ADsafe [27]. This was impossible with the trivial object system in prior work [16], and thus used an intermediate version of the system described here. However, this work was incomplete at the time of that publication, so that published result had two weaknesses that this work addresses:
  1. We were unable to type an important function, reject_name, which requires the pattern-handling we demonstrate in this paper.[16]
  2. More subtly but perhaps equally important, the types we used in that verification had to hard-code collections of member names, and as such were not future-proof. In the rapidly changing world of browsers, where implementations are constantly adding new operations against which sandbox authors must remain vigilant, it is critical to instead have pattern-based whitelists and blacklists, as shown here.

Despite the use of sophisticated patterns, our type-checker is fast. It runs various examples in approximately one second on an Intel Core i5 processor laptop; even ADsafe verifies in under twenty seconds. Therefore, despite being a prototype tool, it is still practical enough to run on real systems.

6. Related Work

Our work builds on the long history of semantics and types for objects and recent work on semantics and types for scripting languages.

Semantics of Scripting Languages There are semantics for JavaScript, Ruby, and Python that model each language in detail. This paper focuses on objects, eliding many other features and details of individual scripting languages. We discuss the account of objects in various scripting semantics below.

Furr, et al. [14] tackle the complexity of Ruby programs by desugaring to an intermediate language (RIL); we follow the same approach. RIL is not accompanied by a formal semantics, but its syntax suggests a class-based semantics, unlike the record-based objects of λ_S^{ob}. Smeding's Python semantics [30] details multiple inheritance, which our semantics elides. However, it also omits certain reflective operators (e.g., hasattr) that we do model in λ_S^{ob}. Maffeis, et al. [22] account for JavaScript objects in detail. However, their semantics is for an abstract machine, unlike our syntactic semantics. Our semantics is closest to the JavaScript semantics of Guha, et al. [15]. Not only do we omit unnecessary details, but we also abstract the plethora of string-matching operators and

[14] Our actual type-checker is a functor over a signature of patterns.

[15] `https://github.com/brownplt/strobe/tree/master/tests/typable` and `https://github.com/brownplt/strobe/tree/master/tests/strobe-typable` have examples of objects as arrays, field-presence checks, and more.

[16] Typing reject_name requires more than string patterns to capture the numeric checks and the combination of predicates, but string patterns let us reason about the charAt checks that it requires.

make object membership checking manifest. These features were not presented in their work, but buried in their implementation of desugaring.

Extensible Records The representation of objects in λ_S^{ob} is derived from extensible records, surveyed by Fisher and Mitchell [12] and Bruce, et al. [7]. We share similarities with ML-Art [29], which also types records with explicitly absent members. Unlike these systems, member names in λ_S^{ob} are first-class strings; λ_S^{ob} includes operators to enumerate over members and test for the presence of members. First-class member names also force us to deal with a notion of infinite-sized patterns in member names, which existing object systems don't address. ML-Art has a notion of "all the rest" of the members, but we tackle types with an arbitrary number of such patterns.

Nishimura [25] presents a type system for an object calculus where messages can be dynamically selected. In that system, the *kind* of a dynamic message specifies the finite set of messages it may resolve to at runtime. In contrast, our string types can describe potentially-infinite sets of member names. This generalization is necessary to type-check the programs in this paper where objects' members are dynamically computed from strings (section 2). Our object types can also specify a wider class of invariants with presence annotations, which allow us to also type-check common scripting patterns that employ reflection.

Types and Contracts for Untyped Languages There are various type systems retrofitted onto untyped languages. Those that support objects are discussed below.

Strongtalk [6] is a typed dialect of Smalltalk that uses protocols to describe objects. String patterns can describe more ad hoc objects than the protocols of Strongtalk, which are a finite enumeration of fixed names. Strongtalk protocols may include a brand; they are thus a mix of nominal and structural types. In contrast, our types are purely structural, though we do not anticipate any difficulty incorporating brands.

Our work shares features with various JavaScript type systems. In Anderson, et al. [5]'s type system, objects' members may be potentially present; it employs strong updates to turn these into definitely present members. Recency types [17] are more flexible, support member type-changes during initialization, and account for additional features such as prototypes. Zhao's type system [36] also allows unrestricted object extension, but omits prototypes. In contrast to these works, our object types do not support strong updates via mutation. We instead allow possibly-absent members to turn into definitely-present members via member presence checking, which they do not support. Strong updates are useful for typing initialization patterns. In these type systems, member names are first-order labels. Thiemann's [32] type system for JavaScript allows first-class strings as member names, which we generalize to member patterns.

RPython [3] compiles Python programs to efficient byte-code for the CLI and the JVM. Dynamically updating Python objects cannot be compiled. Thus, RPython stages evaluation into an interpreted initialization phase, where dynamic features are permitted, and a compiled running phase, where dynamic features are disallowed. Our types introduce no such staging restrictions.

DRuby [13] does not account for member presence checking in general. However, as a special case, An, et al. [2] build a type-checker for Rails-based Web applications that partially-evaluates dynamic operations, producing a program that DRuby can verify. In contrast, our types tackle membership presence testing directly.

System D [8] uses dependent refinements to type dynamic dictionaries. System D is a purely functional language, whereas λ_S^{ob} also accounts for inheritance and state. The authors of System D suggest integrating a string decision procedure to reason about dictionary keys. We use DPRLE [20] to support exactly this style of reasoning.

Heidegger, et al. [18] present *dynamically*-checked contracts for JavaScript that use regular expressions to describe objects. Our implementation uses regular expressions for *static* checking.

Extensions to Class-based Objects In scripting languages, the shape on an object is not fully determined by its class (section 4.1). Our object types are therefore structural, but an alternative class-based system would require additional features to admit scripts. For example, Unity adds structural constraints to Java's classes [23]; class-based reasoning is employed by scripting languages but not fully investigated in this paper. Expanders in eJava [35] allow classes to be augmented, affecting all objects; scripting also allows individual objects to be customized, which structural typing admits. $\mathcal{F}$ickle allows objects to change their class at runtime [4]; our types do admit class-changing (assigning to `"parent"`), but they do not have a direct notion of class. J& allows packages of related classes to be composed [26]. The scripting languages we consider do not support the runtime semantics of J&, but do support related mechanisms such as mixins, which we can easily model and type.

Regular Expression Types Regular tree types and regular expressions can describe the structure of XML documents (e.g., XDuce [21]) and strings (e.g., XPerl [31]). These languages verify XML-manipulating and string-processing programs. Our type system uses patterns not to describe trees of objects like XDuce, but to describe objects' member names. Our string patterns thus allow individual objects to have semi-determinate shapes. Like XPerl, member names are simply strings, but our strings are used to index objects, which are not modeled by XPerl.

7. Conclusion

We present a semantics for objects with first-class member names, which are a characteristic feature of popular scripting languages. In these languages, objects' member names are first-class strings. A program can dynamically construct new names and reflect on the dynamic set of member names in an object. We show by example how programmers use first-class member names to build several frameworks, such as ADsafe (JavaScript), Ruby on Rails, Django (Python), and even Java Beans. Unfortunately, existing type systems cannot type-check programs that use first-class member names. Even in a typed language, such as Java, misusing first-class member names causes runtime errors.

We present a type system in which well-typed programs do not signal "member not found" errors, even when they use first-class member names. Our type system uses string patterns to describe sets of members names and presence annotations to describe their position on the inheritance chain. We parameterize the type system over the representation of patterns. We only require that pattern containment is decidable and that patterns are closed over union, intersection, and negation. Our implementation represents patterns as regular expressions and (co-)finite sets, seamlessly converting between the two.

Our work leaves several problems open. Our object calculus models a common core of scripting languages. Individual scripting languages have additional features that deserve consideration. The dynamic semantics of some features, such as getters, setters, and eval, has been investigated [28]. We leave investigating more of these features (such as proxies), and extending our type system to account for them, as future work. We validate our work with an implementation for JavaScript, but have not built type-checkers for other scripting languages. A more fruitful endeavor might be to build a common type-checking backend for all these languages, if it is possible to reconcile their differences.

Acknowledgments

We thank Cormac Flanagan for his extensive feedback on earlier drafts. We are grateful to StackOverflow for unflagging attention to detail, and to Claudiu Saftoiu for serving as our low-latency interface to it. Gilad Bracha helped us understand the history of Smalltalk objects and their type systems. We thank William Cook for useful discussions and Michael Greenberg for types. Ben Lerner drove our Coq proof effort. We are grateful for support to the US National Science Foundation.

References

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

[2] J. D. An, A. Chaudhuri, and J. S. Foster. Static typing for Ruby on Rails. In *IEEE International Symposium on Automated Software Engineering*, 2009.

[3] D. Ancona, M. Ancona, A. Cuni, and N. D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *ACM SIGPLAN Dynamic Languages Symposium*, 2007.

[4] D. Ancona, C. Anderson, F. Damiani, S. Drossopoulou, P. Giannini, and E. Zucca. A provenly correct translation of Fickle into Java. *ACM Transactions on Programming Languages and Systems*, 2, 2007.

[5] C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *European Conference on Object-Oriented Programming*, 2005.

[6] G. Bracha and D. Griswold. Strongtalk: Typechecking Smalltalk in a production environment. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 1993.

[7] K. B. Bruce, L. Cardelli, and B. C. Pierce. Comparing object encodings. *Information and Computation*, 155(1–2), 1999.

[8] R. Chugh, P. M. Rondon, and R. Jhala. Nested refinements for dynamic languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[9] W. R. Cook, W. L. Hill, and P. S. Canning. Inheritance is not subtyping. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1990.

[10] D. Crockford. ADSafe. `www.adsafe.org`, 2011.

[11] T. V. Cutsem and M. S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In *ACM SIGPLAN Dynamic Languages Symposium*, 2010.

[12] K. Fisher and J. C. Mitchell. The development of type systems for object-oriented languages. *Theory and Practice of Object Systems*, 1, 1995.

[13] M. Furr, J. D. An, J. S. Foster, and M. Hicks. Static type inference for Ruby. In *ACM Symposium on Applied Computing*, 2009.

[14] M. Furr, J. D. A. An, J. S. Foster, and M. Hicks. The Ruby Intermediate Language. In *ACM SIGPLAN Dynamic Languages Symposium*, 2009.

[15] A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming*, 2010.

[16] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming*, 2011.

[17] P. Heidegger and P. Thiemann. Recency types for dynamically-typed, object-based languages: Strong updates for JavaScript. In *ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, 2009.

[18] P. Heidegger, A. Bieniusa, and P. Thiemann. Access permission contracts for scripting languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[19] P. Hooimeijer and M. Veanes. An evaluation of automata algorithms for string analysis. In *International Conference on Verification, Model Checking, and Abstract Interpretation*, 2011.

[20] P. Hooimeijer and W. Weimer. A decision procedure for subset constraints over regular languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[21] H. Hosoya, J. Vouillon, and B. C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27, 2005.

[22] S. Maffeis, J. C. Mitchell, and A. Taly. An operational semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems*, 2008.

[23] D. Malayeri and J. Aldrich. Integrating nominal and structural subtyping. In *European Conference on Object-Oriented Programming*, 2008.

[24] M. S. Miller, M. Samuel, B. Laurie, I. Awad, and M. Stay. Caja: Safe active content in sanitized JavaScript. Technical report, Google, Inc., 2008. `http://google-caja.googlecode.com/files/caja-spec-2008-06-07.pdf`.

[25] S. Nishimura. Static typing for dynamic messages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1998.

[26] N. Nystrom, X. Qi, and A. C. Myers. J&: Nested intersection for scalable software composition. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 2006.

[27] J. G. Politz, S. A. Eliopoulos, A. Guha, and S. Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security Symposium*, 2011.

[28] J. G. Politz, M. J. Carroll, B. S. Lerner, and S. Krishnamurthi. A tested semantics for getters, setters, and eval in JavaScript. In *ACM SIGPLAN Dynamic Languages Symposium*, 2012.

[29] D. Rémy. Programming objects with ML-ART: an extension to ML with abstract and record types. In M. Hagiya and J. C. Mitchell, editors, *Theoretical Aspects of Computer Software*, volume 789 of *Lecture Notes in Computer Science*. Springer-Verlag, 1994.

[30] G. J. Smeding. An executable operational semantics for Python. Master's thesis, Utrecht University, 2009.

[31] N. Tabuchi, E. Sumii, and A. Yonezawa. Regular expression types for strings in a text processing language. *Electronic Notes in Theoretical Computer Science*, 75, 2003.

[32] P. Thiemann. Towards a type system for analyzing JavaScript programs. In *European Symposium on Programming*, 2005.

[33] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.

[34] S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ACM SIGPLAN International Conference on Functional Programming*, 2010.

[35] A. Warth, M. Stanojević, and T. Millstein. Statically scoped object adaption with expanders. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 2006.

[36] T. Zhao. Type inference for scripting languages with implicit extension. In *ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, 2010.