Semantics and Types for Safe Web Programming

by

Arjun Guha

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2012

This dissertation by Arjun Guha is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date ______________ ______________________________

Shriram Krishnamurthi, Director

Recommended to the Graduate Council

Date ______________ ______________________________

Matthias Felleisen, Reader
Northeastern University

Date ______________ ______________________________

Steven P. Reiss, Reader
Brown University

Date ______________ ______________________________

Cormac Flanagan, Reader
University of California, Santa Cruz

Approved by the Graduate Council

Date ______________ ______________________________

Peter M. Weber
Dean of the Graduate School

Acknowledgements

This dissertation is the work of many.

My advisor, Shriram, has been incredibly patient in his training and always had more faith in my abilities than myself. This work started as Claudiu Saftoiu's undergraduate research project, which clearly grew out of control. We had a wonderful time discovering the true depth of his project together. Shortly after, Joe Politz pushed our semantics and type-checker until it could respectably tackle object-oriented programs and security problems. But, we could not have completed our work without Doug Crockford's support and insight. During a particularly trying time, Joe observed that "Some decisions in life are so important, like picking a career or a spouse, but you can't really iterate until you make the right decision, so you just have to do it". He thus inspired me to press on.

This dissertation draws heavily from Matthias Felleisen's work, style, and his invaluable feedback over the years. It would not have been possible without his support. I've made frequent and even unannounced visits to Northeastern University. My conversations with Dave Herman, Sam Tobin-Hochstadt, and the rest of NEU PRL have been invaluable. Our type-checker for JavaScript borrows heavily from Sam's pioneering work on Typed Racket.

Cormac Flanagan and I have had wonderful, helpful conversations about this work, and I look forward to many more. Steve Reiss provided invaluable feedback and very carefully read my drafts. Robby Findler and Casey Klein have always answered my novice questions on PLT Redex, which was an instrumental tool for this work.

I had two wonderful breaks at Google Research and Microsoft Research during the course of my work. At Google, Mark Lentczner was simply the best manager, and also taught me how to test code. I had several illuminating conversations on varied topics with Mark Miller; I'd like to thank him for his support, and for introducing me to all his capabilities. At Microsoft Research, Nikhil Swamy greatly informed my perspective of type theory, which is central to this dissertation, and

Ben Livshits taught me how to focus on essential problems.

Early on, the Flapjax project introduced me to JavaScript, Leo Meyerovich, and Michael Greenberg. Leo reliably provides helpful provocation, and Michael always answers my arcane type theory questions over Google Chat. In addition, I'd like to thank Michael for his judgment.

I first joined Brown PLT as an undergraduate intern, where Guillaume Marceau introduced me to macrology. That summer, Spiros Eliopoulos, who shared my fascinations and enthusiasm, convinced me to come to Brown. My introduction to programming languages happened at Grinnell College, under the guidance of Sam Rebelsky and John Stone.

Since I was terribly bored working on the last bits of this dissertation, Andrew Ferguson let me work on his instead. I look forward to many more fun projects in the future. I'm similarly thrilled that Saurabh Das, my fellow QBASIC hacker, is hacking again.

This past year, the Brown PLT office has been a wonderful place to work. We've grown from a two-student operation to include Andrew Ferguson, Ben Lerner, Betsy Hilliard, Danny Yoo, Hannah Quay-de la Vallee, Joe the Shark, and other guests and animals. The faculty, staff, and other students have been very tolerant and even supportive of our antics.

Ma and Buba always encouraged my years spent in study, while Peen still provides endless entertainment. I've abused the hospitality of Phyo, Htike Htike, Davis, and Tiffany several times while at Brown. For example, I visited the Kyaw Soe Ba Kyu hotel ten times last year. These escapes made graduate school significantly easier.

This dissertation is dedicated to Ammi, Dadu, and Thamma, whose principles will always guide my work.

Contents

List of Figures

Chapter 1

Introduction

Thesis Statement JavaScript programs use prototype-based objects, flow-based type reasoning and other techniques that confound existing type systems. We can design a practical type system that admits the features and idioms of third-party JavaScript programs that were not written with a type-checker in mind. We can integrate a type system with dataflow analysis in a principled manner to account for flow-based type reasoning.

1.1 Why JavaScript?

JavaScript is the *lingua franca* of the Web. Programs written in JavaScript are fundamentally distinct from traditional programs in two key ways. First, applications *freely compose code from several sources.* Second, the user often *cannot control which programs run*; Web programs are visited, not installed. Social networks, such as Facebook, are an exemplar of this behavior. Not only do third-party apps and games embed themselves in Facebook, but Facebook embeds itself in many Web pages. For example, to integrate with Facebook, the The New York Times runs an amalgamation of its own code and Facebook's code (in addition to code from advertising networks). In essence, The New York Times' readers must visit Facebook.

JavaScript and the DOM—the language and libraries of the Web—lack modularity mechanisms needed to safely compose programs. Moreover, JavaScript has many quirks and mis-features that make it difficult to write and reason about even simple snippets of code. As Web programs grow larger and more complex, programmers need tools to reason about JavaScript.

1.2 Overview of Contributions

This dissertation presents a type system for JavaScript and demonstrates that it allows programmers to reason about Web programs. Specifically, we type-check a collection of third-party programs, making few refactorings and finding various bugs (chapter 7). In addition, we use our type-checker find bugs in and verify ADsafe, a security-critical JavaScript framework (chapter 8).

This dissertation presents two novel type-checking techniques to tackle idiomatic JavaScript programs that confound existing techniques. First, we observe that JavaScript programmers use control and state to reason about the types of variables. We present *flow typing*, a method by which a traditional type-checker uses information from a dataflow analysis in a simple and modular way to account for stateful and flow-sensitive reasoning (chapter 4). Second, we observe that objects in JavaScript have arbitrary and dynamically constructed fields. We present *fluid object types* to account for these dynamic objects (chapter 5).

In the tradition of programming languages research, the type systems and program analyses above are accompanied by proofs of soundness. However, such proofs require a dynamic semantics for JavaScript. This dissertation therefore presents λ_{JS}, a core calculus for JavaScript (chapter 2). We employ λ_{JS} in all our technical work and in our implemented tools. Unlike most core calculi, λ_{JS} is accompanied by an implemented desugaring function that translates JavaScript programs to λ_{JS} programs. Furthermore, *we test our semantics* and achieve bug-compatibility with real implementations on a portion of a third-party JavaScript test suite. This gives us confidence that our tools and theorems have some bearing on reality. Testing is particular important for security; malicious programs attack implementations, not theorems.

This dissertation focuses on JavaScript, but many of the ideas are applicable to other scripting languages, such as Ruby, Python, and Lua. We discuss how the ideas in our semantics and our type systems may be employed to build systems for these other languages (chapter 9).

Chapter 2

Semantics

JavaScript, due to its pervasiveness, is a popular research target. There are many frameworks, tools, and sub-languages of JavaScript (e.g, [15, 18, 25, 37, 38, 51, 66]) that tackle security and other problems in JavaScript-based Web applications. These works do not demonstrate soundness, partly because they lack a tractable semantics of JavaScript. The JavaScript standard [22] is capacious and informal while one major formal semantics [58] is large, not amenable to conventional proof techniques, and inherits the standard's complexities, as we discuss in section 2.4. These prior semantics leave unanswered some basic questions. (Such as: Is JavaScript lexically scoped?) More significantly, the sheer size of these semantics makes them unsuitable for building tools and doing detailed proofs. We need a *tractable* semantics for JavaScript.

In this chapter:[1]

- We present a core calculus, λ_{JS}, that embodies JavaScript's essential features, excluding `eval`. λ_{JS} fits on three pages and lends itself well to proof techniques such as subject reduction. We exploit these properties in subsequent chapters.

- We show that we can desugar JavaScript to λ_{JS}. In particular, desugaring handles some of JavaScript's most notorious features, such as `this` and `with`, so λ_{JS} itself remains simple (and thus simplifies proofs that utilize it).

- We mechanize both λ_{JS} and desugaring.

[1]This chapter is based on joint work with Claudiu Saftoiu [40].

- To show compliance with reality, we successfully test λ_{JS} and desugaring against a portion of the actual Mozilla JavaScript test suite.

2.1 λ_{JS}: A Tractable Semantics for JavaScript

JavaScript is full of surprises. Syntax that may have a conventional interpretation for many readers often has a subtly different semantics in JavaScript. To aid the reader, we introduce λ_{JS} incrementally. We include examples of JavaScript's quirks and show how λ_{JS} faithfully models them.

Figures 2.1, 2.2, 2.4, 2.8 and 2.9 specify the syntax and semantics of λ_{JS}. We use a Felleisen-Hieb small-step operational semantics with evaluation contexts [26]. Throughout this dissertation, we typeset λ_{JS} code in a sans-serif typeface, and JavaScript in a `fixed-width typeface`.

2.1.1 Functions, Objects and State

We begin with the small subset of λ_{JS} specified in fig. 2.1 that includes just functions and objects. We model operations on objects via functional update. This seemingly trivial fragment already exhibits some of JavaScript's quirks:

- In field lookup, the name of the field need not be specified statically; instead, field names may be computed at runtime (E-GetField):

```
let (obj = { "x" : 500, "y" : 100 })
  let (select = func(name). obj[name])
    select("x") + select("y")
↪* 600
```

- A program that looks up a non-existent field does not result in an error; instead, JavaScript returns the value **undefined** (E-GetField-NotFound):

```
{ "x" : 7 }["y"] ↪ undefined
```

- Field update in JavaScript is conventional (E-UpdateField)—

```
{ "x" : 0 }["x"] = 10 ↪ { "x" : 10 }
```

  —but the same syntax also creates new fields (E-CreateField):

```
{ "x" : 0 }["z"] = 20 ↪ {"z" : 20, "x" : 10 }
```

$$
\begin{array}{rcl}
c & = & num \mid str \mid bool \mid \textbf{undefined} \mid \textbf{null} \\
v & = & c \mid \textbf{func}(x \cdots)\texttt{.}e \mid \texttt{\{}\ str\texttt{:}v \cdots\ \texttt{\}} \\
e & = & x \mid v \mid \textbf{let}\ (x\ \texttt{=}\ e)\ e \mid e(e\cdots) \mid e\texttt{[}e\texttt{]} \mid e\texttt{[}e\texttt{]}\ \texttt{=}\ e \mid \textbf{delete}\ e\texttt{[}e\texttt{]} \\
E & = & \bullet \mid \textbf{let}\ (x\ \texttt{=}\ E)\ e \mid E(e\cdots) \mid v(v\cdots\ E,\ e\cdots) \\
 & \mid & \texttt{\{}str\texttt{:}\ v\cdots\ str\texttt{:}E,\ str\texttt{:}e\cdots\ \texttt{\}} \mid E\texttt{[}e\texttt{]} \mid v\texttt{[}E\texttt{]} \mid E\texttt{[}e\texttt{]}\ \texttt{=}\ e \mid v\texttt{[}E\texttt{]}\ \texttt{=}\ e \\
 & \mid & v\texttt{[}v\texttt{]}\ \texttt{=}\ E \mid \textbf{delete}\ E\texttt{[}e\texttt{]} \mid \textbf{delete}\ v\texttt{[}E\texttt{]}
\end{array}
$$

$$\textbf{let}\ (x\ \texttt{=}\ v)\ e \hookrightarrow e[x/v] \qquad \text{(E-Let)}$$

$$(\textbf{func}(x_1\cdots x_n)\texttt{.}e)(v_1\cdots v_n) \hookrightarrow e[x_1/v_1\cdots x_n/v_n] \qquad \text{(E-App)}$$

$$\texttt{\{}\ \cdots str\texttt{:}\ v\cdots\ \texttt{\}[}str\texttt{]} \hookrightarrow \text{v} \qquad \text{(E-GetField)}$$

$$\frac{str_x \notin (str_1\cdots str_n)}{\texttt{\{}\ str_1\texttt{:}\ v_1\ \cdots\ str_n\texttt{:}\ v_n\ \texttt{\}}\ \texttt{[}str_x\texttt{]} \hookrightarrow \textbf{undefined}} \qquad \text{(E-GetField-NotFound)}$$

$$\begin{array}{l}\texttt{\{}\ str_1\texttt{:}\ v_1\cdots\ str_i\texttt{:}\ v_i\ \cdots str_n\texttt{:}\ v_n\ \texttt{\}}\ \texttt{[}str_i\texttt{]}\ \texttt{=}\ \text{v} \\ \quad\hookrightarrow \texttt{\{}\ str_1\texttt{:}\ v_1\cdots\ str_i\texttt{:}\ v\ \cdots str_n\texttt{:}\ v_n\ \texttt{\}}\end{array} \qquad \text{(E-UpdateField)}$$

$$\frac{str_x \notin (str_1\cdots)}{\texttt{\{}\ str_1\texttt{:}\ v_1\cdots\ \texttt{\}}\ \texttt{[}str_x\texttt{]}\ \texttt{=}\ v_x \hookrightarrow \texttt{\{}\ str_x\texttt{:}\ v_x\texttt{,}\ str_1\texttt{:}\ v_1\cdots\ \texttt{\}}} \qquad \text{(E-CreateField)}$$

$$\begin{array}{l}\textbf{delete}\ \texttt{\{}\ str_1\texttt{:}\ v_1\cdots\ str_x\texttt{:}\ v_x\ \cdots str_n\texttt{:}\ v_n\ \texttt{\}}\ \texttt{[}str_x\texttt{]} \\ \quad\hookrightarrow \texttt{\{}\ str_1\texttt{:}\ v_1\cdots str_n\texttt{:}\ v_n\ \texttt{\}}\end{array} \qquad \text{(E-DeleteField)}$$

$$\frac{str_x \notin (str_1\cdots)}{\textbf{delete}\ \texttt{\{}\ str_1\texttt{:}\ v_1\cdots\ \texttt{\}}\ \texttt{[}str_x\texttt{]} \hookrightarrow \texttt{\{}\ str_1\texttt{:}\ v_1\cdots\ \texttt{\}}} \qquad \text{(E-DeleteField-NotFound)}$$

Figure 2.1: Functions and Objects

- Finally, JavaScript lets us delete fields from objects:

  delete `{ "x": 7, "y": 13}["x"]` $\hookrightarrow$ `{ "y": 13 }`

JavaScript also supports a more conventional dotted-field notation: `obj.x` is valid JavaScript, and is equivalent to `obj["x"]`. To keep λ_{JS} small, we omit the dotted-field notation in favor of the more general computed lookup, and instead explicitly treat dotted fields as syntactic sugar.

Assignment and Imperative Objects

JavaScript has two forms of state: objects are mutable, and variables are assignable. We model both variables and imperative objects with first-class mutable references (fig. 2.2).[2] We desugar

[2]In the semantics, we use $E\langle e\rangle$ instead of the conventional $E[e]$ to denote a filled evaluation context, to avoid confusion with JavaScript's objects.

$$\begin{array}{rcll} l & = & \cdots & \text{Locations} \\ v & = & \cdots \mid l & \text{Values} \\ \sigma & = & (l, v) \cdots & \text{Stores} \\ e & = & \cdots \mid e \texttt{ = } e \mid \textsf{\textbf{ref}}\ e \mid \textsf{\textbf{deref}}\ e & \text{Expressions} \\ E & = & \cdots \mid E \texttt{ = } e \mid v \texttt{ = } E \mid \textsf{\textbf{ref}}\ E \mid \textsf{\textbf{deref}}\ E & \text{Evaluation Contexts} \end{array}$$

$$\frac{e_1 \hookrightarrow e_2}{\sigma E\langle e_1\rangle \rightarrow \sigma E\langle e_2\rangle}$$

$$\frac{l \notin dom(\sigma) \qquad \sigma' = \sigma, (l, v)}{\sigma E\langle \textsf{\textbf{ref}}\ v\rangle \rightarrow \sigma' E\langle l\rangle} \tag{E-Ref}$$

$$\sigma E\langle \textsf{\textbf{deref}}\ l\rangle \rightarrow \sigma E\langle \sigma(l)\rangle \tag{E-Deref}$$

$$\sigma E\langle l \texttt{ = } v\rangle \rightarrow \sigma[l := v] E\langle v\rangle,\ \text{if } l \in dom(\sigma) \tag{E-SetRef}$$

We use $\twoheadrightarrow$ to denote the reflexive-transitive closure of $\rightarrow$.

Figure 2.2: Mutable References in λ_{JS}

```
function sum(arr) {
  var r = 0;
  for (var i = 0; i < arr["length"]; i = i + 1) {
    r = r + arr[i] };
  return r };

sum([1,2,3]) ↠ 6
var a = [1,2,3,4];
delete a["3"];
sum(a) ↠ NaN
```

Figure 2.3: Array Processing in JavaScript

JavaScript to explicitly allocate and dereference heap-allocated values in λ_{JS}.

Example: JavaScript Arrays JavaScript has arrays that developers tend to use in a traditional imperative style. However, JavaScript arrays are really objects, and this can lead to unexpected behavior. Figure 2.3 shows a small example of a seemingly conventional use of arrays. Deleting the field `a["3"]` (E-DeleteField) does not affect `a["length"]` or shift the array elements. Therefore, in the loop body, `arr["3"]` evaluates to `undefined`, via E-GetField-NotFound. Finally, adding `undefined` to a number yields `NaN`; we discuss other quirks of addition in section 2.1.6.

$$\frac{str_x \notin (str_1 \cdots str_n) \qquad \text{"\_\_proto\_\_"} \notin (str_1 \cdots str_n)}{\texttt{\{ } str_1 \texttt{ : } v_1 \texttt{ , } \cdots \texttt{ , } str_n \texttt{ : } v_n \texttt{ \} [}str_x\texttt{]} \hookrightarrow \textbf{undefined}} \text{(E-GetField-NotFound)}$$

$$\frac{str_x \notin (str_1 \cdots str_n)}{\texttt{\{ } str_1 \texttt{ : } v_1 \cdots \text{"\_\_proto\_\_"}\texttt{: } \textbf{null} \cdots str_n \texttt{ : } v_n \texttt{ \} [}str_x\texttt{]} \hookrightarrow \textbf{undefined}} \text{(E-GetField-Proto-Null)}$$

$$\frac{str_x \notin (str_1 \cdots str_n)}{\texttt{\{ } str_1 \texttt{ : } v_1 \cdots \text{"\_\_proto\_\_"}\texttt{: } l \cdots str_n \texttt{ : } v_n \texttt{ \} [}str_x\texttt{]} \hookrightarrow \texttt{(}\textbf{deref } l\texttt{)[}str_x\texttt{]}} \text{(E-GetField-Proto)}$$

Figure 2.4: Prototype-Based Objects

2.1.2 Prototype-Based Objects

JavaScript supports *prototype inheritance* [11]. For example, in the following code, `animal` is the prototype of `dog`:

```
var animal = { "length": 13, "width": 7 };
var dog = { "__proto__": animal, "barks": true };
```

Prototypes affect field lookup:

```
dog["length"] ↠ 13
dog["width"] ↠ 7

var lab = { "__proto__": dog, "length": 2 }
lab["length"] ↠ 2
lab["width"] ↠ 7
lab["barks"] ↠ true
```

But, prototype inheritance does not affect field update. The code below creates the field `dog["width"]`, but it does not affect `animal["width"]`, which `dog` had previously inherited:

```
dog["width"] = 19
dog["width"] ↠ 19
animal["width"] ↠ 7
```

However, `lab` now inherits `dog["width"]`:

```
lab["width"] ↠ 19
```

Figure 2.4 specifies prototype inheritance. The figure modifies E-GetField-NotFound to only apply when the "__proto__" field is missing.

$desugar[\![$`{prop: e ···}`$]\!] =$

```
ref {
  prop : desugar⟦e⟧ ···,
  "__proto__": (deref Object)["prototype"]
}
```

$desugar[\![$`function(x ···) { stmt ··· }`$]\!] =$

```
 ref {
  "code": func(this, x ···) { return desugar⟦stmt ···⟧ },
  "prototype": ref { "__proto___": (deref Object)["prototype"] } }
```

$desugar[\![$`new e_f(e ···)}`$]\!] =$

```
  let (constr = deref desugar⟦e_f⟧)
    let (obj = ref { "__proto__" : constr["prototype"]})
      constr["code"](obj, desugar⟦e⟧ ···);
      obj
```

$desugar[\![$`obj[field](e ···)`$]\!] =$

```
  let (obj = desugar⟦obj⟧)
    let (f = (deref obj)[field])
      f["code"](obj, desugar⟦e⟧ ···)
```

$desugar[\![$`e_f(e ···)`$]\!] =$

```
  let (obj = desugar⟦e_f⟧)
    let (f = deref obj)
      f["code"](window, desugar⟦e⟧ ···)
```

$desugar[\![$`obj instanceof constr`$]\!] =$

```
  let (obj = ref (deref desugar⟦obj⟧),
       constr = deref desugar⟦constr⟧)
    done: {
      while (deref obj !== null) {
        if ((deref obj)["__proto__"] === constr["prototype"]) {
          break done true }
        else { obj = (deref obj)["__proto__"] } };
      false }
```

$desugar[\![$`this`$]\!] =$ `this` (an ordinary identifier, bound by functions)
$desugar[\![e.x]\!] =$ $desugar[\![e]\!]$`["`x`"]`

Figure 2.5: Desugaring JavaScript's Object Syntax

Prototype inheritance is simple, but it is obfuscated by JavaScript's syntax. The examples in this section are not standard JavaScript because the `"__proto__"` field is not directly accessible by JavaScript programs.[3] In the next section, we unravel and desugar JavaScript's syntax for prototypes.

2.1.3 Prototypes

JavaScript programmers can indirectly manipulate prototypes using syntax that is reminiscent of class-based languages like Java. In this section, we explain this syntax and its actual semantics. We

[3]Some browsers, such as Firefox, can run these examples.

```
var obj = {
  "x" : 0,
  "setX": function(val) { this.x = val } };

// window is the name of the global object in Web browsers
window.x ↠ undefined
obj.setX(10);
obj.x ↠ 10
var f = obj.setX;
f(90);
obj.x ↠ 10 // obj.x was not updated
window.x ↠ 90 // window.x was created
```

Figure 2.6: Implicit `this` Parameter

account for this class-like syntax by desugaring it to manipulate prototypes directly (section 2.1.2). Therefore, this section does not grow λ_{JS} and only describes desugaring. Figure 2.5 specifies the portion of desugaring that is relevant for the rest of this section.

The `this` Keyword

JavaScript does not have conventional methods. Function-valued fields are informally called "methods", and provide an interpretation for a `this` keyword, but both are quite different from those of, say, Java.

For example, in fig. 2.6, when `obj.setX(10)` is applied, `this` is bound to `obj` in the body of the function. In the same figure however, although `f` is bound to `obj.setX`, `f(90)` does not behave like a traditional method call. In fact, the function is applied with `this` bound to the *global object* [22, Section 10.1.5].

In general, `this` is an implicit parameter to all JavaScript functions. Its value is determined by the syntactic shape of function applications. Thus, when we desugar functions to λ_{JS}, we make `this` an explicit argument. Moreover, we desugar function calls to explicitly supply a value for `this`.

Functions as Objects

In JavaScript, functions are objects with fields:

```
f = function(x) { return x + 1 }

f.y = 90

f(f.y) ↠ 91
```

We desugar JavaScript's `function` to objects in $\lambda_{JS}$with a distinguished `code` field that refers to the actual function. Therefore, we also desugar application to lookup the `code` field.

We could design λ_{JS} so that functions truly are objects, making this bit of desugaring unnecessary. In our experience, JavaScript functions are rarely used as objects. Therefore, our design lets us reason about simple functions when possible, and functions as objects only when necessary.

In addition to the `code` field, which we add by desugaring, and any other fields that may have been created by the programmer, all functions also have a distinguished field called `prototype`. As fig. 2.5 shows, the `prototype` field is a reference to an object that eventually leads to the prototype of `Object`. Unlike the `__proto__` field, `prototype` is accessible and can be updated by programmers. The combination of its mutability and its use in `instanceof` leads to unpredictable behavior, as we show below.

Constructors and Prototypes

JavaScript does not have explicit constructors, but it does have a `new` keyword that invokes a function with `this` bound to a new object. For example, the following code—

```
function Point(x, y) {
  this.x = x;
  this.y = y }

pt = new Point(50, 100)
```

—applies the function `Point` and returns the value of `this`. `Point` explicitly sets `this.x` and `this.y`. Moreover, `new Point` implicitly sets `this.__proto__` to `Point.prototype`. We can now observe prototype inheritance:

```
Point.prototype.getX = function() { return this.x }
pt.getX() ↠ pt.__proto__.getX() ↠ 50
```

In standard JavaScript, because the `__proto__` field is not exposed, the only way to set up a prototype hierarchy is to update the `prototype` field of functions that are used as constructors.

The `instanceof` Operator

JavaScript's `instanceof` operator has an unconventional semantics that reflects the peculiar notion of constructors that we have already discussed. In most languages, a programmer might expect that if `x` is bound to the value created by `new Constr(···)`, then `x instanceof Constr` is true. In JavaScript, however, this invariant does not apply.

```
function Dog() { this.barks = "woof" };
function Cat() { this.purrs = "meow" };
dog = new Dog();
cat = new Cat();
dog.barks; ↠ "woof"
cat.purrs; ↠ "meow"

function animalThing(obj) {
  if (obj instanceof Cat) { return obj.purrs }
  else if (obj instanceof Dog) { return obj.barks }
  else { return "unknown animal" } };

animalThing(dog); ↠ "woof"
animalThing(cat); ↠ "meow"
animalThing(4234); ↠ "unknown animal"

Cat.prototype = Dog.prototype;
animalThing(cat); ↠ "unknown animal"
animalThing(dog) ↠ undefined // dog.purrs (E-GetField-NotFound)
```

Figure 2.7: Using `instanceof`

For example, in fig. 2.7, `animalThing` dispatches on the type of its argument using `instanceof`. However, after we set `Cat.prototype = Dog.prototype`, the type structure seems to break down. The resulting behavior might appear unintuitive in JavaScript, but it is straightforward when we desugar `instanceof` into λ_{JS}. In essence, `cat instanceof Cat` is `cat.__proto__ === Cat.prototype`.[4] In the figure, before `Cat.prototype = Dog.prototype` is evaluated, the following are true:

`cat.__proto__ === Cat.prototype`

`dog.__proto__ === Dog.prototype`

`Cat.prototype !== Dog.prototype`

However, after we update `Cat.prototype`, we have:

`cat.__proto__ ===` the previous value of `Cat.prototype`

`dog.__proto__ === Dog.prototype`

`Cat.prototype === Dog.prototype`

Hence, `cat instanceof Cat` becomes `false`. Furthermore, since `animalThing` first tests for `Cat`, the test `dog instanceof Cat` succeeds.

2.1.4 Statements and Control Operators

JavaScript has a plethora of control statements. Many map directly to λ_{JS}'s control operators (fig. 2.8), while the rest are easily desugared.

[4]The `===` operator is the physical equality operator, akin to `eq?` in Scheme.

$$
\begin{array}{rcl}
label & = & \text{(Labels)} \\
e & = & \cdots \mid \textbf{if } (e)\ e \textbf{ else } e \mid e;e \mid \textbf{while}(e)\ e \mid label\texttt{:}\ e \\
 & \mid & \textbf{break } label\ e \mid \textbf{try } e \textbf{ catch } (x)\ e \mid \textbf{try } e \textbf{ finally } e \\
 & \mid & \textbf{err } v \mid \textbf{throw } e \\
E & = & \cdots \mid \textbf{if } (E)\ e \textbf{ else } e \mid E;e \mid label\texttt{:}\ E \mid \textbf{break } label\ E \\
 & \mid & \textbf{try } E \textbf{ catch } (x)\ e \mid \textbf{try } E \textbf{ finally } e \mid \textbf{throw } E \\
E' & = & \bullet \mid \textbf{let } (x \texttt{ = } v \cdots x \texttt{ = } E'\texttt{,}\ x \texttt{ = } e \cdots)\ e \mid E'(e\cdots) \mid v(v\cdots\ E'\texttt{,}\ e\cdots) \\
 & \mid & \textbf{if } (E')\ e \textbf{ else } e \mid \{\ str\texttt{:}\ v\cdots\ str\texttt{:}\ E'\texttt{,}\ str\texttt{:}\ e\cdots\ \} \\
 & \mid & E'[e] \mid v[E'] \mid E'[e] \texttt{ = } e \mid v[E'] \texttt{ = } e \mid v[v] \texttt{ = } E' \mid E' \texttt{ = } e \mid v \texttt{ = } E' \\
 & \mid & \textbf{delete } E'[e] \mid \textbf{delete } v[E'] \mid \textbf{ref } E' \mid \textbf{deref } E' \mid E';\ e \mid \textbf{throw } E' \\
F & = & E' \mid label\texttt{:}\ F \mid \textbf{break } label\ F \quad \text{(Exception Contexts)} \\
G & = & E' \mid \textbf{try } G \textbf{ catch } (x)\ e \quad \text{(Local Jump Contexts)}
\end{array}
$$

$$\textbf{if } (\textbf{true})\ e_1 \textbf{ else } e_2 \hookrightarrow e_1 \qquad \text{(E-IfTrue)}$$

$$\textbf{if } (\textbf{false})\ e_1 \textbf{ else } e_2 \hookrightarrow e_2 \qquad \text{(E-IfFalse)}$$

$$v;e \hookrightarrow e \qquad \text{(E-Begin-Discard)}$$

$$\textbf{while}(e_1)\ e_2 \hookrightarrow \textbf{if } (e_1)\ e_2;\ \textbf{while}(e_1)\ e_2 \textbf{ else undefined} \qquad \text{(E-While)}$$

$$\textbf{throw } v \hookrightarrow \textbf{err } v \qquad \text{(E-Throw)}$$

$$\textbf{try } F\langle\textbf{err } v\rangle \textbf{ catch } (x)\ e \hookrightarrow e[x/v] \qquad \text{(E-Catch)}$$

$$\sigma F\langle\textbf{err } v\rangle \to \sigma\textbf{err } v \qquad \text{(E-Uncaught-Exception)}$$

$$\textbf{try } F\langle\textbf{err } v\rangle \textbf{ finally } e \hookrightarrow e;\ \textbf{err } v \qquad \text{(E-Finally-Error)}$$

$$\textbf{try } G\langle\textbf{break } label\ v\rangle \textbf{ finally } e \hookrightarrow e;\ \textbf{break } label\ v \qquad \text{(E-Finally-Break)}$$

$$\textbf{try } v \textbf{ catch } (x)\ e \hookrightarrow v \qquad \text{(E-Catch-Pop)}$$

$$\textbf{try } v \textbf{ finally } e \hookrightarrow e;\ v \qquad \text{(E-Finally-Pop)}$$

$$label\texttt{:}\ G\langle\textbf{break } label\ v\rangle \hookrightarrow v \qquad \text{(E-Break)}$$

$$\frac{label_1 \neq label_2}{label_1\texttt{:}\ G\langle\textbf{break } label_2\ v\rangle \hookrightarrow \textbf{break } label_2\ v} \qquad \text{(E-Break-Pop)}$$

$$label\texttt{:}\ v \hookrightarrow v \qquad \text{(E-Label-Pop)}$$

$$\textbf{break } label_1\ G\langle\textbf{break } label_2\ v\rangle \hookrightarrow \textbf{break } label_2\ v \qquad \text{(E-Break-Break)}$$

Figure 2.8: Control operators for λ_{JS}

For example, consider JavaScript's `return` and `break` statements. A `break` l statement transfers control to the local label l. A `return` e statement transfers control to the end of the local function and produces the value of e as the result. Instead of two control operators that are almost identical, λ_{JS} has a single **break** expression that produces a value.

Concretely, we elaborate JavaScript's functions to an expression with a label ret:

$$desugar[\![\texttt{function(}x\cdots\texttt{) \{ } stmt\cdots \texttt{ \} }]\!] = \textsf{\textbf{func}}\texttt{(}this\ \ x\cdots\texttt{).(}ret\texttt{: } desugar[\![stmt\cdots]\!]\texttt{)}$$

Thus, `return` statements are desugared to **break** ret:

$$desugar[\![\texttt{return } e]\!] = \textsf{\textbf{break}}\ \ \textsf{ret}\ \ desugar[\![e]\!]$$

while `break` statements are desugared to produce **undefined**:

$$desugar[\![\texttt{break } label]\!] = \textsf{\textbf{break}}\ \ label\ \ \textsf{\textbf{undefined}}$$

2.1.5 Static Scope in JavaScript

The JavaScript standard specifies identifier lookup in an unconventional manner. It uses neither substitution nor environments, but *scope objects* [22, Section 10.1.4]. A scope object is akin to an activation record, but is a conventional JavaScript object. The fields of this object are interpreted as variable bindings.

In addition, a scope object has a distinguished parent-field that references another scope object. (The global scope object's parent-field is `null`.) This linked list of scope objects is called a *scope chain*. The value of an identifier `x` is the value of the first `x`-field in the *current scope chain*. When a new variable `y` is defined, the field `y` is added to the scope object at the head of the scope chain.

Since scope objects are ordinary JavaScript objects, JavaScript's `with` statement lets us add arbitrary objects to the scope chain. Given the features discussed below, which include `with`, it is not clear whether JavaScript is lexically scoped. In this section, we describe how JavaScript's scope-manipulation statements are desugared into λ_{JS}, which is obviously lexically scoped.

Local Variables

In JavaScript, functions close over their current scope chain (intuitively, their static environment). Applying a closure sets the current scope chain to be that in the closure. In addition, an empty scope object is added to the head of the scope chain. The function's arguments and local variables (introduced using `var`) are properties of this scope object.

Local variables are automatically *lifted* to the top of the function. As a result, in a fragment such as this—

```
function foo() {
  if (true) { var x = 10 }
  return x }

foo() ↠ 10
```

—the `return` statement has access to the variable that appears to be defined inside a branch of the `if`. This can result in somewhat unintuitive answers:

```
function bar(x) {
  return function() {
    var x = x;
    return x }}

bar(200)() ↠ undefined
```

Above, the programmer might expect the `x` on the right-hand side of `var x = x` to reference the argument `x`. However, due to lifting, all bound occurrences of `x` in the nested function reference the local variable `x`. Hence, `var x = x` reads and writes back the initial value of `x`. The initial value of local variables is `undefined`.

We can easily give a lexical account of this behavior. A local variable declaration, `var x = e`, is desugared to an assignment, `x = e`. Furthermore, we add a let-binding at the top of the enclosing function:

let (x = **ref undefined**) $\cdots$

Global Variables

Global variables are subtle. Global variables are properties of the global scope object (`window`), which has a field that references itself:

```
window.window === window ↠ true
```

Therefore, a program can obtain a reference to the global scope object by simply referencing `window`.[5]

[5]In addition, `this` is bound to `window` in function applications (fig. 2.5).

As a consequence, globals seem to break lexical scope, since we can observe that they are properties of `window`:

```
var x = 0;
window.x = 50;
x ↠ 50
x = 100;
window.x ↠ 100
```

However, `window` is the only scope object that is directly accessible to JavaScript programs [22, Section 10.1.6]. We maintain lexical scope by abandoning global variables. That is, we simply desugar the obtuse code above to the following:

```
window.x = 0;
window.x = 50;
window.x ↠ 50
window.x = 100;
window.x ↠ 100
```

Although global variables observably manipulate `window`, local variables are still lexically scoped. We can thus reason about local variables using substitution, α-renaming, and other standard techniques.

With Statements

The `with` statement is a widely-acknowledged JavaScript wart.[6] A `with` statement adds an arbitrary object to the front of the scope chain:

```
function(x, obj) {
  with(obj) {
    x = 50; // if obj.x exists, then obj.x = 50, else x = 50
    return y } } // similarly, return either obj.y, or window.y
```

We can desugar `with` by turning the comments above into code:

```
function(x, obj) {
  if (obj.hasOwnProperty("x")) { obj.x = 50 }
  else { x = 50 }
  if ("y" in obj) { return obj.y }
  else { return window.y } }
```

[6] Indeed, the latest revision of the JavaScript specification [23] has a *strict mode* that eliminates some of JavaScript's most offensive misfeatures, including `with`.

$$\begin{array}{rcl} e & = & \cdots \mid op_n\texttt{(}e_1 \cdots e_n\texttt{)} \\ E & = & \cdots \mid op_n\texttt{(}v \cdots E\ e \cdots\texttt{)} \\ E' & = & \cdots \mid op_n\texttt{(}v \cdots E'e \cdots\texttt{)} \\ \delta_n & : & op_n \times v_1 \cdots v_n \rightarrow c + err \end{array}$$

$$op_n\texttt{(}v_1 \cdots v_n\texttt{)} \hookrightarrow \delta_n(op_n, v_1 \cdots v_n) \qquad \text{(E-Prim)}$$

Figure 2.9: Primitive Operators

Nested `with`s require a little more care, but can be dealt with in the same manner. However, desugaring `with` is non-compositional. We will return to this point in section 3.4.

What are Scope Objects? Various authors (including ourselves) have developed JavaScript tools that work with a subset of JavaScript that is intuitively lexically scoped [5, 18, 25, 38, 43, 66]. We show how JavaScript can be desugared into lexically scoped λ_{JS}, validating these assumptions. As a result, we no longer need scope objects in the specification; they may instead be viewed as an implementation strategy.[7]

2.1.6 Type Conversions and Primitive Operators

JavaScript is not a pure object language. We can observe the difference between primitive numbers and number objects:

```
x = 10;
y = new Number(7)
typeof x ↠ "number"
typeof y ↠ "object"
```

Moreover, JavaScript's operators include implicit type conversions between primitives and corresponding objects:

```
x + y ↠ 17
```

We can redefine these type conversions without changing objects' values:

```
Number.prototype.valueOf = function() { return 0 }
x + y ↠ 10
y.toString() ↠ "7"
```

[7]Scope objects are especially well suited for implementing `with`. Our desugaring strategy for `with` increases code-size linearly in the number of nested `with`s, which scope-objects avoid.

Both + and * perform implicit coercions, and + also concatenates strings:

```
x + y.toString() ↠ "107" // 10 converted to the string "10"
x * y.toString() ↠ 70 // "7" converted to the number 7
```

This suggests that JavaScript's operators are complicated. Indeed, the standard specifies `x + y` with a 15-step algorithm [22, Section 11.6.1] that refers to three pages of metafunctions. Buried in these details are four primitive operators: primitive addition, string concatenation, and number-to-string and string-to-number type coercions.

These four primitives are essential and intuitive. We therefore model them with a conventional δ function (fig. 2.9). The remaining details of operators are type-tests and method invocations; as the examples above suggest, JavaScript internally performs operations such as `y.valueOf()` and `typeof x`. In λ_{JS} we make these type-tests and method calls explicit.

This chapter does not enumerate all the primitives that λ_{JS} needs. Instead, the type of δ constrains their behavior significantly, which often lets us reason without a specific δ function. (For instance, due to the type of δ, we know that primitives cannot manipulate the heap.)

2.2 Soundness and Adequacy of λ_{JS}

Soundness We mechanize λ_{JS} with PLT Redex [26]. The process of mechanizing helped us find errors in our semantics, particularly in the interactions of control operators (fig. 2.8). We use our mechanized semantics to test [56] λ_{JS} for safety. Note that we do not prove this property, since a progress proof for an untyped language is straightforward. We believe that PLT Redex's randomized testing is sufficient.

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

- $e \in v$,
- $e = \textsf{err}\ v$, *for some* v, *or*
- $\sigma e \rightarrow \sigma' e'$, *where* $\sigma' e'$ *is a closed, well-formed configuration.*

This property requires additional evaluation rules for runtime type errors, and definitions of well-formedness. We elide them as they are conventional.

$$\begin{tikzcd}
\text{JavaScript Program} \arrow[r, "desugar"] \arrow[d, "\text{Real Implementations}"'] & \lambda_{JS}\ \text{Program} \arrow[d, "\lambda_{JS}\ \text{interpreter}"] \\
\texttt{stdout} \arrow[r, leftrightarrow, "\texttt{diff}"'] & \texttt{stdout}
\end{tikzcd}$$

Figure 2.10: Testing Strategy for λ_{JS}

Syntactic Form	Occurrences (approx.)
`with` blocks	15
`var` statements	500
`try` blocks	20
`function`s	200
`if` and `switch` statements	90
`typeof` and `instanceof`	35
`new` expressions	50
`Math` library functions	15

Figure 2.11: Test Suite Coverage

Adequacy λ_{JS} is a semantics for the core of JavaScript. We have described how it models many aspects of the language's semantics, warts and all. Ultimately, however, a small core language has limited value to those who want to reason about programs written in full JavaScript.

Given our method of handling JavaScript via desugaring, we are obliged to show that desugaring and the semantics enjoy two properties. First, we must show that all JavaScript programs can be desugared to λ_{JS}.

Claim 1 (Desugaring is Total) *For all eval-free JavaScript programs e, desugar*$[\![e]\!]$ *is defined.*

Second, we must demonstrate that our semantics corresponds to what JavaScript implementations actually do.

Claim 2 (Desugared Code Produces the Same Output) *For all* `eval`*-free JavaScript programs, e, desugar*$[\![eval_{JavaScript}(e)]\!]$ *produces the same output as* $eval_{\lambda_{JS}}(desugar[\![e]\!])$.

We could try to prove these claims, but that just begs the question:

What is $eval_{JavaScript}$?

A direct semantics would require evidence of its own adequacy. In practice, JavaScript is truly defined by its major implementations. Open-source Web browsers are accompanied by extensive JavaScript test suites. These test suites help the tacit standardization of JavaScript across major implementations.[8] We use these test suites to *test* our semantics.

Figure 2.10 outlines our testing strategy. We first define an interpreter for λ_{JS}. This is a straightforward exercise; the interpreter is a mere 100 LOC, and easy to inspect since it is based directly on the semantics.[9] Then, for any JavaScript program, we should be able to run it both directly and in our semantics.[10] For direct execution we employ three JavaScript implementations: SpiderMonkey (used by Firefox), V8 (used by Chrome), and Rhino (an implementation in Java). We desugar the same program into λ_{JS} and run the result through our interpreter. We then check whether our λ_{JS} interpreter produces the same output as each JavaScript implementation.

2.3 Conclusion

Our tests cases are a significant portion of the Mozilla JavaScript test suite. We omit the following tests:

- Those that target Firefox-specific JavaScript extensions.
- Those that use `eval`.
- Those that target library details, such as regular expressions.

The remaining tests are about 5,000 LOC unmodified.

Our λ_{JS} interpreter produces exactly the same output as Rhino, V8, and SpiderMonkey on the entire test suite. Figure 2.11 indicates that these tests employ many interesting syntactic forms, including statements like `with` and `switch` that are considered complicated. We make the following observations:

- No prior semantics for JavaScript accounts for all these forms (e.g., Maffeis et al. [58] do not model `switch`).
- We account for much of JavaScript by desugaring. Therefore, these tests validate both our core semantics and our desugaring strategy.
- These tests give us confidence that our implemented tools are correct.

[8]For example, the Firefox JavaScript test suite is also found in the Safari source.
[9]PLT Redex can evaluate expressions in a mechanized semantics. However, our tests are too large for Redex's evaluator.
[10]To observe output, we add a primitive operator printing operator to the δ function.

2.4 Related Work

JavaScript Semantics JavaScript is specified in 200 pages of prose and pseudocode [22]. This specification is barely amenable to informal study, let alone proofs. Maffeis, Mitchell, and Taly [58] present a 30-page operational semantics, based directly on the JavaScript specification. Their semantics covers most of JavaScript directly, but does omit a few syntactic forms.

Our approach is drastically different. λ_{JS} is a semantics for the core of JavaScript, though we desugar the rest of JavaScript into λ_{JS}. In section 2.2, we present evidence that our strategy is correct. λ_{JS} and desugaring together are much smaller and simpler than the semantics presented by Maffeis, et al. Yet, we cover all of JavaScript (other than `eval`) and account for a substantial portion of the standard libraries as well.

Maffeis, Mitchell, and Taly [58] define an operational semantics for JavaScript that closely follows the JavaScript specification. In contrast, our semantics is factored as a core calculus and a translation to the core. We believe this factoring makes it easier to build tools and proofs for our semantics. An further difference is that we demonstrate faithfulness to implementations by desugaring and running third-party tests in our semantics.

A technical advantage of our semantics is that it is conventional. For example, we use substitution instead of scope objects (section 2.1.5). Therefore, we can use conventional techniques, such as subject reduction, to reason in λ_{JS}. It is unclear how to build type systems for a semantics that uses scope objects.

David Herman [46] defines a CEKS machine for a small portion of JavaScript. This machine is also based on the standard and inherits some of its complexities, such as implicit type conversions.

CoreScript [93] models an imperative subset of JavaScript, along with portions of the DOM, but omits essentials such as functions and objects. Moreover, their big-step semantics is not easily amenable to typical type safety proofs.

Object Calculi λ_{JS} is an untyped, object-based language with prototype inheritance. However, λ_{JS} does not have methods as defined in object calculi. Without methods, most object calculi cease to be interesting. However, we do desugar JavaScript's method invocation syntax to self-application in λ_{JS} [1, Chapter 18].

λ_{JS} and JavaScript do not support cloning, which is a crucial element of other prototype-based languages, such as Self [88]. JavaScript does support Self's prototype inheritance, but the surface syntax of JavaScript does not permit direct access to an object's prototype (section 2.1.3). Without cloning, and without direct access to the prototype, JavaScript programmers cannot use techniques such as dynamic inheritance and mode-switching [1].

Chapter 3

Verifying a Simple Web Sandbox

This chapter[1] is a simple application of λ_{JS} (chapter 2) to verify a commonly employed language-based Web sandboxing technique. In chapter 8, we verify an actual third-party Web sandbox and consider its high-level design. This chapter details some of the programming language techniques employed in chapter 8 in a simple setting.

3.1 Isolating Untrusted Code

Web platforms often combine programs from several different sources on the same page. For instance, on a portal like iGoogle, a user can build a page that combines a weather widget with a stock ticker widget; on Facebook, users can run applications. Unfortunately, this means distinct programs can interfere with each other, which creates the possibility that a malicious application may steal data or cause other harm. To prevent both accidents and malice, sites must somehow sandbox widgets.

To this end, platform developers have defined safe sub-languages (often called "safe subsets") of JavaScript like ADsafe [18], Caja [66], and Facebook JavaScript (FBJS) [25]. These are designed as sub-languages of JavaScript to target developers who already know how to write JavaScript Web applications. These sub-languages disallow blatantly dangerous features such as `eval`. However, they also try to establish more subtle security properties using syntactic restrictions, as well as runtime checks that they insert into untrusted code. Naturally, this raises the question whether these sub-languages function as advertised.

[1]This chapter is based on joint work with Claudiu Saftoiu [40].

Let us consider the following property, which is inspired by FBJS and Caja: we wish to prevent code in the sandbox from communicating with a server. For instance, we intend to block the XMLHttpRequest object:

```
var x = new window.XMLHttpRequest()
x.open("GET", "/get_confidential", false)
x.send("");
var result = x.responseText
```

For simplicity, we construct a sub-language that just disallows access to XMLHttpRequest. A complete solution would use our techniques to block other communication mechanisms, such as `document.write` and `Element.innerHTML`.

We begin with short, type-based proofs that exploit the compactness of λ_{JS}. We then use our tools to migrate from λ_{JS} to JavaScript.

3.2 Isolating JavaScript

"Disallow access to XMLHttpRequest" is ambiguous and must be precisely defined. In JavaScript, `window.XMLHttpRequest` references the XMLHttpRequest constructor, where `window` names the global object. We make two assumptions:

- In λ_{JS}, we allocate the global object at location `0`. This is a convenient convention that is easily ensured by desugaring.
- The XMLHttpRequest constructor is only accessible as a property of the global object. This assumption is valid as long as we do not use untrusted libraries (or can analyze their code).

Given these two assumptions, we can formally state "disallow access to XMLHttpRequest" as a property of λ_{JS} programs:

Definition 1 (Security) *e is safe if* $e \neq E\langle\langle$ **deref** (**ref** *0*)$\rangle$ *["XMLHttpRequest"]*$\rangle$.

Note that in the definition above, the active expression is (**deref** (**ref** 0)), and the evaluation context is $E\langle\bullet$["XMLHttpRequest"]$\rangle$.

Intuitively, ensuring safety appears to be easy. Given an untrusted λ_{JS} program, we can elaborate property accesses, e_1[e_2], to *lookup*(e_1,e_2), where *lookup* is defined in fig. 3.1.

```
lookup = func(obj, field) {
  return if (field === "XMLHttpRequest") { undefined }
         else { (deref obj)[field] }
}
```

Figure 3.1: Safe Wrapper for λ_{JS}

$T = \mathbf{JS}$

$$\Gamma \vdash \mathit{string} : \mathbf{JS} \tag{T-String}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T} \tag{T-Id}$$

$$\frac{\Gamma, x_1 : \mathbf{JS}, \cdots, x_n : \mathbf{JS} \vdash e : \mathbf{JS}}{\Gamma \vdash \texttt{func } (x_1 \cdots x_n)\ \texttt{\{ return e \}} : \mathbf{JS}} \tag{T-Fun}$$

$$\frac{\Gamma \vdash e_1 : \mathbf{JS} \qquad \cdots \qquad \Gamma \vdash e_n : \mathbf{JS}}{\Gamma \vdash \delta_n(op_n, e_1 \cdots e_n) : \mathbf{JS}} \tag{T-Prim}$$

The type judgments for remaining forms are similar to T-Prim and T-Fun: namely, $\Gamma \vdash e : \mathbf{JS}$ if all subexpressions of e have type **JS**. However, $e_1[e_2]$ is *not typable*.

Figure 3.2: Type System that Disallows Field Lookup

This technique[2] has two problems. First, it blocks access to the "XMLHttpRequest" property of *any* object. Second, although *lookup* may appear "obviously correct", the actual wrapping in Caja, FBJS, and other sub-languages occurs in JavaScript, not in a core calculus like λ_{JS}. Hence, *lookup* does not directly correspond to any JavaScript function. We could write a JavaScript function that resembles *lookup*, but it would be wrought with various implicit type conversions and method calls (section 2.1.6) that could break its intended behavior. Thus, we start with safety for λ_{JS} before tackling JavaScript's details.

3.3 Types for Securing λ_{JS}

Our goal is to determine whether a λ_{JS} program is safe (definition 1). We wish to do so without making unnecessary assumptions. In particular, we do not assume that *lookup* (fig. 3.1) is itself safe.

We begin by statically disallowing *all* field accesses. The trivial type system in fig. 3.2 achieves this, since it excludes a typing rule for $e_1[e_2]$. This type system does not catch conventional type errors. Instead, it has a single type, **JS**, of statically safe JavaScript expressions (definition 1). The

[2]Maffeis et al.'s blacklisting [59], based on techniques used in FBJS, has this form.

$T = \cdots \mid \textbf{NotXHR}$

$$\textbf{NotXHR} <: \textbf{JS} \qquad \text{(Sub-Safe)}$$

$$\frac{\Gamma \vdash e : S \qquad S <: T}{\Gamma \vdash e : T} \qquad \text{(T-Sub)}$$

$$\frac{v \neq \text{"XMLHttpRequest"}}{\Gamma \vdash v : \textbf{NotXHR}} \qquad \text{(T-SafeValue)}$$

$$\frac{\Gamma \vdash e_1 : \textbf{JS} \qquad \Gamma \vdash e_2 : \textbf{NotXHR}}{\Gamma \vdash e_1\texttt{[}e_2\texttt{]} : \textbf{JS}} \qquad \text{(T-GetField)}$$

$$\frac{x \in dom(\Gamma) \qquad \Gamma \vdash e_2 : \textbf{JS} \qquad \Gamma[x : \textbf{NotXHR}] \vdash e_3 : \textbf{JS}}{\Gamma \vdash \textbf{if } \texttt{(}x \texttt{ === } \text{"XMLHttpRequest"}\texttt{) \{ } e_2 \texttt{ \} } \textbf{else} \texttt{ \{ } e_3 \texttt{ \}} : \textbf{JS}} \qquad \text{(T-IfSafe)}$$

Figure 3.3: Type System for Blocking Access to XMLHttpRequest

$$\frac{\Gamma \vdash e_2 : \textbf{JS}}{\Gamma \vdash \textbf{if } \texttt{(}\text{"XMLHttpRequest"}\texttt{=== }\text{"XMLHttpRequest"}\texttt{) \{ } e_2 \texttt{ \} } \textbf{else} \texttt{ \{ } e_3 \texttt{ \}} : \textbf{JS}} \qquad \text{(T-IfTrue-XHR)}$$

$$\frac{\Gamma \vdash e_2 : \textbf{JS}}{\Gamma \vdash \textbf{if } \texttt{(}\textbf{true}\texttt{) \{ } e_2 \texttt{ \} } \textbf{else} \texttt{ \{ } e_3 \texttt{ \}} : \textbf{JS}} \qquad \text{(T-IfTrue)}$$

Figure 3.4: Auxiliary Typing Rules for Blocking Access to XMLHttpRequest

following theorem is evidently true:

Theorem 1 *For all λ_{JS} expressions e, if $\cdot \vdash e : T$ and $e \twoheadrightarrow e'$ then e' is safe.*

We need to extend our type system to account for *lookup*, taking care not to violate theorem 1. Note that *lookup* is currently untypable, since field access is untypable. However, the conditional in *lookup* seems to ensure safety; our goal is to prove that it does. Our revised type system is shown in fig. 3.3. The new type, **NotXHR**, is for expressions that provably do not evaluate to the string "XMLHttpRequest". Since primitives like string concatenation yield values of type **JS** (T-Prim in fig. 3.2), programs cannot manufacture unsafe strings with type **NotXHR**. (Of course, trusted primitives could yield values of type **NotXHR**.)

Note this important peculiarity: *These new typing rules are purpose-built for lookup.* There are other ways to establish safe access to fields. However, since we will rewrite all expressions $e_1\texttt{[}e_2\texttt{]}$ to $lookup\texttt{(}e_1\texttt{,}e_2\texttt{)}$, our type system need only account for the syntactic structure of *lookup*.

Our revised type system admits *lookup*, but we must prove theorem 1. It is sufficient to prove the following lemmas:

Lemma 1 (Safety) *If* $\cdot \vdash e : \boldsymbol{JS}$*, then* $e \neq E\langle$`v["XMLHttpRequest"]`$\rangle$*, for any value* v*.*

The proof of this lemma is by induction on typing derivations, given the typing rules in fig. 3.2 and fig. 3.3. This lemma also holds for the typing rules in fig. 3.4, which we introduce below.

Lemma 2 (Preservation) *If* $\cdot \vdash e : \boldsymbol{JS}$*, and* $e \rightarrow e'$*, then* $\cdot \vdash e' : \boldsymbol{JS}$*.*

Proof Technique The typing rules for *lookup* (fig. 3.3) require a technique introduced in *occurrence typing* for Typed Scheme [86].

Although *lookup* is typable, subject reduction requires all expressions in this reduction sequence to be typable:

```
lookup(window, "XMLHttpRequest")
→ if ("XMLHttpRequest" === "XMLHttpRequest") { undefined }
  else { (deref window)["XMLHttpRequest"] }
→ if (true) { undefined }
  else { (deref window)["XMLHttpRequest"] }
→ undefined
```

The intermediate expressions above are not typable, although they are intuitively safe. We can make them typable by extending our type system with the typing rules in fig. 3.4, which let us prove subject reduction.

However, we have to ensure that our new typing rules do not violate safety (lemma 1). Intuitively, lemma 1 still holds, since our newly-typable expressions are not of the form `e["XMLHttpRequest"]`.

Our type system may appear ad hoc, but it simply reflects the nature of JavaScript security solutions. Note that our type system is merely a means to an end: the main result is the conclusion of theorem 1, which is a property of the runtime semantics.

3.4 Scaling to JavaScript

Since we can easily implement a checker for our type system, we might claim we have a result for JavaScript as follows: desugar JavaScript into λ_{JS} and type-check the resultant λ_{JS} code. This strategy is, however, unsatisfying because seemingly harmless changes to a typable JavaScript program may result in a program that fails to type-check, due to the effects of desugaring. This would make the language appear whimsical to the widget developer.

Instead, our goal is to define a safe sub-language (just as, say, Caja and FBJS do). This safe sub-language would provide syntactic safety criteria, such as:

- The JavaScript expression e_1 `+` e_2 is safe when its subexpressions are safe.
- e_1`[`e_2`]`, when rewritten to *lookup*`(`e_1`,` e_2`)`, is safe, but fails if e_2 evaluates to `"XMLHttpRequest"`.

Our plan is as follows. We focus on the *structure* of the desugaring rules and show that a particular kind of compositionality in these rules suffices for showing safety. We illustrate this process by extending the λ_{JS} result to include JavaScript's addition (which, as we explained in section 2.1.6, is non-trivial). We then generalize this process to the rest of the language.

3.4.1 Safety for Addition

By theorem 1, it is sufficient to determine whether $\Gamma \vdash desugar[\![e_1 + e_2]\!] : \mathbf{JS}$. Proving this, however, would benefit from some constraints on e_1 and e_2. Consider the following proposition:

Proposition 1 *If* $\Gamma \vdash desugar[\![e_1]\!] : \boldsymbol{JS}$ *and* $\Gamma \vdash desugar[\![e_2]\!] : \boldsymbol{JS}$*, then* $\Gamma \vdash desugar[\![e_1 + e_2]\!] : \boldsymbol{JS}$*.*

By lemma 1, this proposition entails that if e_1 and e_2 are safe, then $e_1 + e_2$ is safe. But is the proposition true? $desugar[\![e_1 + e_2]\!]$ produces an unwieldy λ_{JS} expression with explicit type-conversions and method calls. Still, a quick inspection of our implementation shows that:

$desugar[\![e_1 + e_2]\!]$ = **let** (x = $desugar[\![e_1]\!]$) **let** (y = $desugar[\![e_2]\!]$) $\cdots$

$desugar[\![e_1 + e_2]\!]$ simply recurs on its subexpressions and does not examine the result of $desugar[\![e_1]\!]$ and $desugar[\![e_2]\!]$. Moreover, the elided body does not contain additional occurrences of $desugar[\![e_1]\!]$ and $desugar[\![e_2]\!]$. Thus, we can write the right-hand side as a two-holed *program context*:

$desugar[\![e_1 + e_2]\!]$ = $C_+\langle desugar[\![e_1]\!], desugar[\![e_2]\!]\rangle$
C_+ = **let** (x = $\bullet_1$) **let** (y = $\bullet_2$) $\cdots$

Therefore, desugaring e_1 `+` e_2 is *compositional.*

A simple replacement lemma [90] holds for our type system:

Lemma 3 (Replacement) *If:*

i. $\mathcal{D}$ *is a deduction concluding* $\Gamma \vdash C[e_1, e_2] : \textbf{\textit{JS}}$*,*

ii. *Subdeductions* $\mathcal{D}_1, \mathcal{D}_2$ *prove that* $\Gamma_1 \vdash e_1 : \textbf{\textit{JS}}$ *and* $\Gamma_2 \vdash e_2 : \textbf{\textit{JS}}$ *respectively,*

iii. $\mathcal{D}_1$ *occurs in* $\mathcal{D}$*, at the position corresponding to* $\bullet_1$*, and* $\mathcal{D}_2$ *at the position corresponding to* $\bullet_2$*, and*

iv. $\Gamma_1 \vdash e'_1 : \textbf{\textit{JS}}$ *and* $\Gamma_2 \vdash e'_2 : \textbf{\textit{JS}}$*,*

then $\Gamma \vdash C\langle e'_1, e'_2\rangle : \textbf{\textit{JS}}$*.*

Replacement, along with weakening of environments, gives us our final lemma:

Lemma 4 *If:*

- $x : \textbf{\textit{JS}}, y : \textbf{\textit{JS}} \vdash C_+[x, y] : \textbf{\textit{JS}}$*, and*
- $\Gamma \vdash desugar[\![e_1]\!] : \textbf{\textit{JS}}$ *and* $\Gamma \vdash desugar[\![e_2]\!] : \textbf{\textit{JS}}$*,*

then $\Gamma \vdash C_+\langle desugar[\![e_1]\!], desugar[\![e_2]\!]\rangle : \textbf{\textit{JS}}$*.*

The conclusion of lemma 4 is the conclusion of proposition 1. The second hypothesis of lemma 4 is the only hypothesis of proposition 1. Therefore, to prove proposition 1, we simply need to prove $x : \mathbf{JS}, y : \mathbf{JS} \vdash C_+\langle x, y\rangle : \mathbf{JS}$.

We establish this using our tools. We assume `x` and `y` are safe (i.e., have type **JS**), and desugar and type-check the expression `x + y`. Because this succeeds, the machinery above—in particular, the replacement lemma—tells us that we may admit `+` into our safe sub-language.

3.4.2 A Safe Sub-Language

The proofs of lemma 3 and 4 do not rely on the definition of C_+. For each construct, we must thus ensure that the desugaring rule can be written as a program context, which we easily verify by inspection. We find this true for all syntactic forms other than `with`, which we omit from our safe sub-language (as do other sub-language such as Caja and FBJS). If `with` were considered important, we could extend our machinery to determine what circumstances, or with what wrapping, it too could be considered safe.

Having checked the structure of the desugaring rules, we must still establish that their expansion does no harm. We mechanically populate a type environment with placeholder variables, create expressions of each kind, and type-check. All forms pass type-checking, except for the following:

- `x[y]` and `x.XMLHttpRequest` do not type—happily, as they are unsafe! This is acceptable because these unsafe forms will be wrapped in *lookup*.
- However, `x[y]++`, `x[y]--`, `++x[y]`, and `--x[y]` also fail to type due to the structure of code they generate on desugaring. Yet, we believe these forms are safe; we could account for them with additional typing rules, as employed below for *lookup*.

3.4.3 Safety for *lookup*

As section 3.3 explained, we designed our type system to account for *lookup* (fig. 3.1). However, *lookup* is in λ_{JS}, whereas we need a wrapper in JavaScript. A direct translation of *lookup* into JavaScript yields:

```
lookupJS = function(obj, field) {
 if (field === "XMLHttpRequest") { return undefined }
 else { return obj[field] } }
```

Since *lookupJS* is a closed expression that is inserted as-is into untrusted scripts, we can desugar and type-check it in isolation. Doing so, however, reveals a surprise: $desugar⟦lookupJS⟧$ does not type-check.

When we examine the generated λ_{JS} code, we see that `obj[field]` is desugared into an expression that explicitly converts `field` to a string. (Recall that field names are strings.) If, however, `field` is itself an object, this conversion includes the method call `field.toString()`. Working backward, we see that the following exploit would succeed:

```
lookupJS(window, { toString: function() { return "XMLHttpRequest" } })
```

where the second argument to *lookupJS* (i.e., the expression in the field position) is a literal object that has a single method, `toString`, which returns `"XMLHttpRequest"`. Thus, not only does *lookupJS* not type, it truly is unsafe!

Our type system successfully caught a bug in our JavaScript implementation of *lookup*. The fix is simple: ensure that `field` is a primitive string:

```
safeLookup = function(obj, field) {
  if (field === "XMLHttpRequest") { return undefined }
  else if (typeof field === "string") { return obj[field] }
  else { return undefined } }
```

This code truly is safe, though to prove it we need to extend our type system. We design the extension by studying the result of desugaring $safeLookup$.[3]

We have noted that desugaring evinces the unsafe method call. However, `toString` is called only if `field` is not a primitive. This conditional is inserted *by desugaring*:

```
if (typeof field === "location") { ... field.toString() ... }
else { field }
```

Thus, the second `if` in $safeLookup$ desugars to:

```
if (typeof field === "string") {
  obj[if (typeof field === "location") { ... field.toString() ... }
      else { field }] }
```

To now reach `field.toString()`, both conditions must hold. Since this cannot happen, the unsafe code block is unreachable.

Recall, however, that we designed our type system for λ_{JS} around the syntactic structure of the lookup guard. With this more complex guard, we must extend our type system to employ if-splitting—which we already used in section 3.3—a second time. As long as our extension does not violate safety (lemma 1) and subject reduction (lemma 2), the arguments in this section still hold.

3.5 Perspective

In the preceding sections, we rigorously developed a safe sub-language of JavaScript that disallows access to XMLHttpRequest. In addition, we outlined a proof of correctness for the runtime "wrapper". To enhance isolation, we have to disallow access to a few other properties, such as `document.write` and `Element.innerHTML`. We could do so with simple variants of the proofs in this chapter.

However, verifying a realistic sub-language of JavaScript requires more effort. Chapter 8 presents our verification of ADsafe, a third-party Web sandbox, which use the main ideas from this chapter:

- We state the safety properties of ADsafe in the same manner as definition 1.
- This chapter used a small, special-purpose type system to verify a safety property. Our ADsafe verification uses a general-purpose JavaScript type system, to tackle the many programming patterns that ADsafe employs.

[3]Desugaring produces 200 LOC of pretty-printed λ_{JS} (appendix F).

  - The type system in this chapter has a single *if-splitting* rule. Chapter 4 demonstrates that JavaScript programs, including ADsafe, use many more control and state dependent patterns to reason about types. The ADsafe verification relies on those techniques.
  - This chapter's type system only ensures that an object's `"XMLHttpRequest"` field is inaccessible. ADsafe enforces many more invariants on objects, some of which cannot be expressed in existing object type systems. Chapter 5 presents an object type system that can express these invariants, which we use to verify ADsafe.

- In section 3.4.2, we used types to derive a safe subset of JavaScript. We derive a safe subset for ADsafe in the same manner. However, we also use tests to ensure our derived safe subset corresponds to JSLint, which is ADsafe's own ad hoc subset.

Our verification of ADsafe also uses λ_{JS}. In fact, the primary goal of this chapter is not to define a safe sub-language of JavaScript, but rather to showcase a simple application of λ_{JS}:

- λ_{JS} is small. It is much smaller than other definitions and semantics for JavaScript. Therefore, our proofs are tractable.

- λ_{JS} is adequate and tested. This gives us confidence that our arguments are applicable to real-world JavaScript.

- λ_{JS} is conventional, so we are free to use standard type-soundness techniques [90]. In contrast, working with JavaScript's scope objects would be onerous. This section is littered with statements of the form $\Gamma \vdash e : \mathbf{JS}$. Heap-allocated scope objects would preclude the straightforward use of Γ, thus complicating the proof effort (and perhaps requiring new techniques).

- Finally, *desugar* is compositional. Although we developed a type system for λ_{JS}, we were able to apply our results to most of JavaScript by exploiting the compositionality of *desugar*.

Chapter 4

Typing Control and State

JavaScript programs employ idioms that confound conventional type systems. In this chapter,[1] we highlight one important set of related idioms: the use of local control and state to reason informally about types. We account for these idioms in two steps. First, we formalize run-time tags and their relationship to types. We then use this relationship to develop a novel strategy that integrates type-checking with flow analysis in a modular way. We demonstrate that in our separation of typing and flow analysis, each component remains conventional, their composition is simple, but the result can handle these idioms better than either one alone.

4.1 Patterns of Control and State

JavaScript (and many other scripting languages) do not support pattern-matching or tagged data constructors. Therefore, programmers have to use reflection to reason about values. For example, the following program uses the `typeof` operator, which returns a string representing the "runtime type" of its argument:[2]

```
/*: Num ∪ {x : Num, y : Num} → Num */
function fromOrigin(p) {
  if (typeof p === "object") {
    return Math.sqrt(p.x * p.x + p.y * p.y);
  }
```

[1] This chapter is based on joint work with Claudiu Saftoiu [41].

[2] To be precise, `typeof` does not return a (static) type but a (runtime) tag. This distinction becomes significant when we retrofit an actual type system onto JavaScript (section 4.3).

```
 0 /*: ⊤ → Str ∪ Bool */
   function serialize(val) {
     switch (typeof val) {
       case "undefined":
       case "function":
 5       return false;
       case "boolean":
         return val ? "true" : "false";
       case "number":
         return "" + val;
10     case "string":
         return val;
     }

     if (val === null) { return "null"; }
15
     var fields /*: [Str] */ = [ ];
     for (var p in val) {
       var v = serialize(val[p]);
       if (typeof v === "string") {
20       fields.push(p + ": " + v);
       }
     }
     return "{ " + fields.join(", ") + " }";
   }
```

Figure 4.1: Non-local control

```
  else {
    return Math.abs(p);
  }
}
```

Such patterns are pervasive and they suggest we need untagged union types, as written in the comment. If we use union types, then we have to account for the `if`-statement above, which narrows the type of `p` to $\{x : \mathsf{Num}, y : \mathsf{Num}\}$ and Num in the respective branches. However, JavaScript has other control operators that are also used to reason about types. We consider more examples before devising a type-checking strategy.

The function in fig. 4.1 serializes arbitrary values to strings ($\top$ is the type of all expressions).[3] Functions and the special value `undefined` cannot be serialized, so for these it returns `false`. Let us informally reason about `serialize` to determine if it is type-safe.

On line 3, the function branches on the result of `typeof val`:

- For `case "undefined"`, control falls through to line 5.
- On line 6, for `case "function"`, the function returns `false`.

[3]This example is based on `toJSON` from the popular Prototype library.

- On line 8, for `case "boolean"`, the function branches on `val` and returns either `"true"` or `"false"`. `val` is a boolean because none of the preceding cases fall through to line 8.
- On line 10, for `case "number"`, the function uses string concatenation to coerce the number `val` to a string. `val` is a number because none of the preceding cases fall through to line 10.
- On line 12, for `case "string"`, the function returns `val`. `val` is a string because none of the preceding cases fall through to here.

This `switch` is missing a case. If `typeof val === "object"`, then none of the cases above will match and control will fall through. However, since all the explicitly handled cases return, we know that `typeof val === "object"` holds on lines 15—24.

JavaScript has a value `null` and `typeof null === "object"`. Therefore, line 15 tests for `null` and if the test is `true`, the program returns `"null"`. However, if the test is `false`, since the conditional does not have a `false`-branch, control proceeds to line 17. Since the `true`-branch returns, `val !== null` holds on lines 17–24. We can safely use `val` as an object on these lines. Lines 20—21 also employ flow-directed reasoning, but are relatively trivial. Therefore, we can conclude that `serialize` is safe.

We reasoned about `serialize` by following its convoluted control-flow instead of merely following its syntactic structure. JavaScript forces such reasoning on programmers. The primary culprit is `return`, which *aborts* control flow and produces a result. Since `serialize` uses multiple `return` statements, our reasoning relies on `return` aborting its local continuation.

Can we rewrite this function in a style that enables syntactic reasoning? Using the ternary operator instead of `if` and `switch` statements, we might try:

```
function syntactic_serialize(val) {
  return (typeof val === "undefined") ? false :
         ...
         (val === "null") ? "null" :
         var fields = []; // syntax error: statement
         for (var p in val) // // syntax error: statement
         ... }
```

Unfortunately, variable bindings and loops are statements, which do not compose with expressions. (This syntactic defect is shared by Ruby, Python, and other scripting languages.) This complicates the rewriting (which, in turn, could negatively impact error reporting), and suggests it would be better to tackle `serialize` directly.

```
/*: [a] ∗ Int ∗ Int ∪ Undef → [a] */
function slice(arr, start, stop) {
  var len = /*: Int */ arr.length;
  if (typeof stop === "undefined") { stop = len - 1; }

  if (start < 0 || stop > len || start > stop) {
    throw "Invalid arguments";
  }

  var r = /*: [a] */ [ ];
  for (var i = 0; i <= stop - start; i++) {
      r[i] = arr[start + i];
  }

  return r;
}
```

Figure 4.2: Heap-Sensitive Reasoning

4.1.1 Heap-Sensitive Reasoning

The function `slice` (fig. 4.2) returns a section of an array:[4]

```
slice([ "A", "B", "C", "D" ], 1, 2) ↠ [ "B", "C" ]
```

However, the third argument (`stop`) is optional:

```
slice([ "A", "B", "C", "D" ], 1) ↠ [ "B", "C", "D" ]
```

JavaScript does not have optional arguments and default parameters. Instead, elided arguments receive the value `undefined` and extraneous arguments are dropped. `slice` relies on this peculiarity to simulate optional arguments.

If `stop` is `undefined`, then the side-effect `stop = len` ensures that `stop` is an integer in the continuation of the `if`-statement. This function relies not only on control-flow, but on the interaction of control and state to reason about types.

Dynamic Dispatch and Type Tests We reasoned about the use of `serialize` and `slice` by following their convoluted control-flow and side-effects, instead of merely following their syntactic structure. A reader may argue that these functions are "bad style", so a type system can legitimately reject them. For example, an easily typable alternative to `serialize` is to extend the builtin prototypes (`Object`, `String`, etc.) with a `serialize` method and rely on dynamic dispatch, instead of reflection. Unfortunately, extending builtin classes runs into the fragile base class problem [65] and is thus considered bad practice (e.g., [35]).

[4]`slice` is part of the JavaScript standard, but is not implemented by some older browsers. This definition is from the 4umi compatibility library.

Perspective The examples above make heavy use of local control and state to reason informally about "types". Section 9.5 shows that such patterns are prevalent in actual JavaScript code (and other scripting languages). A static type system that admits these programs will need to support this style of reasoning and various other features (e.g., objects). The book-keeping needed to account for control and state can pervade the entire type system and occlude its typing of other features.

4.2 Semantics and Types

λ_{JS}, presented in chapter 2 is an adequate model of JavaScript, but contains various details, such as objects, that are orthogonal to our technical presentation. We instead present flow typing using a smaller calculus, λ_S.

Figure 4.3 specifies the syntax and semantics of λ_S, which is a core calculus that is sufficient for our exposition of flow typing. λ_S includes higher-order functions, mutable references, conditionals, a control operator (**break**), and basic primitives. Type annotations (discussed below) are ignored during evaluation.

In this chapter, the static types of λ_S are much richer than its runtime tags. Therefore, we use a more technically precise name, `tagof`, to model the `typeof` operator of real scripting languages. The **break** operator can model both `break` and `return` statements of JavaScript. The **break** operator aborts the current continuation up to a matching label and returns a value. We specify the semantics of three primitives, of which physical equality (`===`) and `tagof` appear extensively in flow-directed reasoning (fig. 4.1). Other expressions, such as `tagof x !== "string"`, are a simple extension of our theory.

Figure 4.3 also specifies the syntax of types, T. Types include untagged unions and a top type $\top$, which were motivated in section 4.1. We also include the type of locations, $\mathsf{Ref}\ T$, and a bottom type $\bot$ for control operators that do not return a value. Given these types, subtyping (fig. 4.4) is conventional.

Our typing relation is also mostly conventional. We present select typing judgments in fig. 4.5. Note that the typing environment binds identifiers and labels. By T-SetRef, we can write subtypes to locations.[5] Finally, like JavaScript, λ_S programs cannot **break** across function boundaries, so we statically disallow it by dropping labels when typing functions (T-Abs).

[5]This is a simple restriction of source and sink types [71, Chapter 15.5].

identifiers	x		
locations	l		
constants	c	$=$	$num \mid str \mid bool \mid$ **undefined**
values	v	$=$	$x \mid c \mid$ **func**$(x \cdots)$:T `{` e `}` $\mid l$
expressions	e	$=$	$v \mid$ **let** x = e_1 in $e_2 \mid e_f(e_1 \cdots e_n) \mid op_n(e_1 \cdots e_n)$
		$\mid$	**if** (e_1) `{` e_2 `}` **else** `{` e_3 `}` $\mid$ **break** $label\ e \mid label$:T `{` e `}`
		$\mid$	**ref** $e \mid$ **deref** $e \mid$ setref $e_1\ e_2$
evaluation contexts	E	$=$	$\bullet \mid$ **let** x = E in $e \mid E(e_1 \cdots e_n) \mid v_f(v \cdots Ee \cdots)$
		$\mid$	$op_n(v \cdots Ee \cdots) \mid$ **break** $label\ E \mid$ **if** (E) `{` e_2 `}` **else** `{` e_3 `}`
		$\mid$	$label$:T `{` E `}` $\mid$ **ref** $E \mid$ **deref** $E \mid$ setref $E\ e \mid$ setref $v\ E$
stores	σ	$=$	$\cdot \mid (l, v)\,\sigma$
types	T	$=$	Str $\mid$ Bool $\mid$ Undef $\mid T_1 \cup T_2 \mid T_1 \cdots \to T \mid$ Ref $T \mid \bot \mid \top$

(E-Let)	$\sigma E\langle \textbf{let}\ x = v\ \text{in}\ e\rangle \to \sigma E\langle e[x/v]\rangle$
(E-Prim)	$\sigma E\langle op_n(v \cdots)\rangle \to \sigma E\langle \delta_n(op_n, v \cdots)\rangle$
(β_v)	$\sigma E\langle \textbf{func}(x \cdots)\ \{\ e\ \}(v \cdots)\rangle \to \sigma E\langle e[x/v \cdots]\rangle$
(E-Break)	$\sigma E_1\langle label{:}\{\ E_2\langle \textbf{break}\ label\ v\rangle\ \}\rangle \to \sigma E_1\langle v\rangle$, when $label \notin E_2$
(E-Label-Pop)	$\sigma E\langle label{:}\{\ v\ \}\rangle \to \sigma E\langle v\rangle$
(E-Ref)	$\sigma E\langle \textbf{ref}\ v\rangle \to (l, v), \sigma E\langle l\rangle$ l fresh
(E-Deref)	$\sigma E\langle \textbf{deref}\ l\rangle \to \sigma E\langle \sigma(l)\rangle$
(E-SetRef)	$\sigma E\langle \text{setref}\ l\ v\rangle \to \sigma[l/v]E\langle l\rangle$

$\delta_1(\text{tagof}, num) =$ "number"	$\delta_2(\texttt{===}, v, v) =$ **true**
$\delta_1(\text{tagof}, \textbf{undefined}) =$ "undefined"	$\delta_2(\texttt{===}, v_1, v_2) =$ **false**, when $v_1 \neq v_2$
$\delta_1(\text{tagof}, str) =$ "string"	$\delta_2(\texttt{-}, num_1, num_2) = num_1 - num_2$
$\delta_1(\text{tagof}, bool) =$ "boolean"	
$\delta_1(\text{tagof}, l) =$ "location"	
$\delta_1(\text{tagof}, \textbf{func}(x \cdots)\ \{\ e\ \}) =$ "function"	

Figure 4.3: Syntax and Semantics of λ_S

$$T <: T \qquad \text{(S-Refl)}$$

$$\frac{S <: U \qquad U <: T}{S <: T} \qquad \text{(S-Trans)}$$

$$\bot <: T \qquad \text{(S-Bot)}$$

$$T <: \top \qquad \text{(S-Top)}$$

$$\frac{S' <: S \cdots \qquad T <: T'}{S \cdots \to T <: S' \cdots \to T'} \qquad \text{(S-Arr)}$$

$$\frac{T <: S \qquad S <: T}{\mathsf{Ref}\ S <: \mathsf{Ref}\ T} \qquad \text{(S-Ref)}$$

$$\frac{S_1 <: T \qquad S_2 <: T}{S_1 \cup S_2 <: T} \qquad \text{(S-UnionE)}$$

$$S <: S \cup T \qquad \text{(S-UnionL)}$$

$$T <: S \cup T \qquad \text{(S-UnionR)}$$

Figure 4.4: Subtyping in λ_S

4.3 Relating Static Types and Runtime Tags

Consider the following JavaScript program:

```
function f(x) {
  if (typeof x === "string") { return 0; }
  else { return (x-1); } }
f(200)
```

We can model this in λ_S as follows, with `x` as a local variable and the **break**s representing `return` statements and the intended type annotation inserted:[6]

```
let f = ref func(y) : Num ∪ Str → Num {
  return:Num {
    let x = ref y in
    if (tagof (deref x) === "string") { break return 0 }
    else { break return ((deref x) - 1) } } }
in (deref f)(200)
```

Both the λ_S and original JavaScript programs run without error, returning `199`.

This λ_S program fails to type in the type checker of the previous section because the `-` (minus) operator expects its operands to be numbers, but **deref** `x` has type $\mathsf{Num} \cup \mathsf{Str}$. However, the tag-test informs us, the reader, that `x` has the static type Str in the true branch; the type annotation on `y` bounds its range of values, and thus enables us to conclude that `x` has type Num in the false branch. Thus, the dynamic test and static type annotation collude to demonstrate that this program is

[6]Chapter 2 desugars JavaScript to λ_{JS} in this form.

$$ty_1(\textsf{tagof}) = \top \to \textsf{Str} \qquad ty_2(\texttt{===}) = \top \times \top \to \textsf{Bool} \qquad ty_2(\texttt{-}) = \textsf{Num} \times \textsf{Num} \to \textsf{Num}$$

$$\frac{\Sigma(l) = T}{\Sigma; \Gamma \vdash l : T} \quad \text{(T-Loc)}$$

$$\frac{\Sigma; \Gamma \vdash e : S \qquad S <: T}{\Sigma; \Gamma \vdash e : T} \quad \text{(T-Sub)}$$

$$\frac{\Sigma; \Gamma', x : S, \cdots \vdash e : T \qquad \Gamma' = \Gamma \text{ with labels removed}}{\Sigma; \Gamma \vdash \textsf{func}(x \cdots) : S \cdots \to T\{\ e\ \} : S \cdots \to T} \quad \text{(T-Abs)}$$

$$\frac{\Sigma; \Gamma \vdash e_1 : \textsf{Ref}\ S \qquad \Sigma; \Gamma \vdash e_2 : T \qquad T <: S}{\Sigma; \Gamma \vdash \textsf{setref}\ e_1\ e_2 : \textsf{Ref}\ T} \quad \text{(T-SetRef)}$$

$$\frac{\Sigma; \Gamma \vdash e_1 : \textsf{Bool} \qquad \Sigma; \Gamma \vdash e_2 : T \qquad \Sigma; \Gamma \vdash e_3 : T}{\Sigma; \Gamma \vdash \textsf{if}\ (e_1)\ \{\ e_2\ \}\ \textsf{else}\ \{\ e_3\ \} : T} \quad \text{(T-If)}$$

$$\frac{\Sigma; \Gamma, label : T \vdash e : T}{\Sigma; \Gamma \vdash label{:}T\ \{\ e\ \} : T} \quad \text{(T-Label)}$$

$$\frac{\Gamma(label) = T \qquad \Sigma, \Gamma \vdash e : T}{\Sigma; \Gamma \vdash \textsf{break}\ label\ e : \bot} \quad \text{(T-Break)}$$

Figure 4.5: Typing λ_S (Essential Rules)

statically safe. Our goal is to enable the static type checker to arrive at the same conclusion.[7]

To support such reasoning, a retrofitted type system must relate static types and runtime tags. We show this in fig. 4.6. *runtime* maps types to tag *sets* (due to the presence of unions), but since types are much richer than tags, we cannot distinguish all static types at runtime, e.g., all arrow types are mapped to the tag "function" (objects would be modeled similarly). *static* lets us narrow a type based on a known tag. For example, if a value has type $\textsf{Str} \cup \textsf{Num}$ and its tag set is {"number"}, then *static* produces the type Num. Note that *static* is partial: for example, *static*({"number"}, Str) is undefined.

Since *static* relates types and tags, our type system can use it to account for runtime tag-tests. We use *static* by extending λ_S with an auxiliary construct, tagcheck R e (fig. 4.7), which narrows the type of e based on the tag set R. By judiciously inserting tagchecks, we can make our example typable.[8] We thus offer tagcheck as an appropriate cast-like operator for scripting languages.

[7]Occurrence typing [86] is a different approach that can also type-check this example.

[8]Section 4.4 presents an efficient technique to insert tagchecks automatically, so they are hidden from the programmer.

$r = \{\text{"string"}, \text{"boolean"}, \text{"number"}, \text{"undefined"}, \text{"function"}, \text{"location"}\}$
$R = \mathcal{P}(r)$

$$\begin{array}{lcl}
\mathit{runtime} : T \to R & & \\
\mathit{runtime}(\mathsf{Str}) & = & \{\text{"string"}\} \\
\mathit{runtime}(\mathsf{Bool}) & = & \{\text{"boolean"}\} \\
\mathit{runtime}(\mathsf{Num}) & = & \{\text{"number"}\} \\
\mathit{runtime}(\mathsf{Undef}) & = & \{\text{"undefined"}\} \\
\mathit{runtime}(S \cup T) & = & \mathit{runtime}(S) \cup \mathit{runtime}(T) \\
\mathit{runtime}(S \cdots \to T) & = & \{\text{"function"}\} \\
\mathit{runtime}(\bot) & = & \emptyset \\
\mathit{runtime}(\top) & = & r \\
\mathit{runtime}(\mathsf{Ref}\ T) & = & \{\text{"location"}\}
\end{array}$$

$$\begin{array}{lcl}
\mathit{static} : R \times T \to T & & \\
\mathit{static}(R, \mathsf{Str}) & = & \mathsf{Str}, \text{if "string"} \in R \\
\mathit{static}(R, \mathsf{Bool}) & = & \mathsf{Bool}, \text{if "boolean"} \in R \\
\mathit{static}(R, \mathsf{Num}) & = & \mathsf{Num}, \text{if "number"} \in R \\
\mathit{static}(R, \mathsf{Undef}) & = & \mathsf{Undef}, \text{if "undefined"} \in R \\
\mathit{static}(R, S \cdots \to T) & = & S \cdots \to T, \text{if "function"} \in R \\
\mathit{static}(R, S \cup T) & = & \mathit{static}(R, S) \cup \mathit{static}(R, T) \\
\mathit{static}(R, S \cup T) & = & \mathit{static}(R, S), \text{if } \mathit{static}(R, T) \text{ is undefined} \\
\mathit{static}(R, S \cup T) & = & \mathit{static}(R, T), \text{if } \mathit{static}(R, S) \text{ is undefined} \\
\mathit{static}(R, \top) & = & \top \\
\mathit{static}(R, \mathsf{Ref}\ T) & = & \mathsf{Ref}\ \top
\end{array}$$

Figure 4.6: Relationship Between Types and Tags

A tagcheck expression can fail in three ways. Two are static: when the tag set R is incompatible with the type of e, *static* is undefined; even if it is compatible, the resulting type may not be what the context expects. However, the third failure is dynamic: if e reduces to v and $\mathsf{tagof}(v) \notin R$, then evaluation gets stuck with a tagerr (E-TagCheck-Err). This error condition manifests itself when we try to prove a type soundness theorem.

The preservation lemma is conventional:

Lemma 5 (Preservation) *If $\Sigma, \cdot \vdash e : T$, $\Sigma \vdash \sigma$, and $\sigma e \to \sigma' e'$, then there exists a Σ', such that:*

i. $\Sigma', \cdot \vdash \sigma' e' : T$,

ii. $\Sigma' \vdash e'$, *and*

iii. $\Sigma \subseteq \Sigma'$.

However, programs can get stuck on tagerrs:

Lemma 6 (Progress) *If $\Sigma, \cdot \vdash e : T$ and $\Sigma \vdash \sigma$, then either:*

$$\begin{array}{rcl} e & = & \cdots \mid \mathsf{tagcheck}\ R\ e \mid \mathsf{tagerr} \\ E & = & \cdots \mid \mathsf{tagcheck}\ R\ E \end{array}$$

$$\frac{\delta_1(\mathsf{tagof}, v) \in R}{\sigma E\langle \mathsf{tagcheck}\ \ R\ \ v\rangle \to \sigma E\langle v\rangle} \qquad \text{(E-TagCheck)}$$

$$\frac{\delta_1(\mathsf{tagof}, v) \notin R}{\sigma E\langle \mathsf{tagcheck}\ \ R\ \ v\rangle \to \sigma E\langle \mathsf{tagerr}\rangle} \qquad \text{(E-TagCheck-Err)}$$

$$\frac{\Sigma; \Gamma \vdash e : S \qquad \mathit{static}(R, S) = T}{\Sigma; \Gamma \vdash \mathsf{tagcheck}\ \ R\ \ e : T} \qquad \text{(T-Check)}$$

$$\Sigma; \Gamma \vdash \mathsf{tagerr} : \bot \qquad \text{(T-TagErr)}$$

Figure 4.7: Typing and Evaluation of Checked Tags

i. $e \in v$, *or*

ii. *there exist* σ' *and* e', *such that* $\sigma e \to \sigma' e'$, *or*

iii. $e = E\langle \mathsf{tagerr}\rangle$, *for some* E.

Thus, the type soundness theorem is unsatisfying because of (iii.) of the lemma above. We could try to "repair" the type system; indeed, a sufficiently complicated type system might not need tagchecks and tagerrs at all. Our key idea is to admit tagerrs to keep the type system simple, and then discharge them by other means.

4.4 Automatically Inserting Safe tagchecks

We need a way to automatically insert tagchecks that fail neither statically nor at runtime. The tagcheck-insertion technique needs to be sound and handle uses of local control and state that we presented in section 4.1. Unlike conventional type systems, flow analyses are well-suited to such reasoning styles, so we consider flow analysis here. Unfortunately, whole-program analysis of functional and object-oriented languages is non-modular and expensive (section 4.6). Moreover, we need to relate abstract heaps produced by flow analysis to types produced by type-checking. We address these problems broadly, before formally presenting one particular analysis (section 4.5).

The goal of the flow analysis is to compute the tag-sets necessary for tagcheck expressions. Therefore, the domain of the analysis will be tag-sets augmented by some book-keeping information.

Returning to the example from section 4.3, the comments illustrate the kind of information we need from flow analysis:

```
0 let f = ref func(y) : Num ∪ Str → Num {
    return:Num { /* tagof(y) ∈ {"number","string"} */
      let x = ref y in /* x = ref y, tagof(y) ∈ {"number","string"} */
      if (tagof (deref x) === "string") { /* same as line 3 */
        break return 0 /* x = ref y, tagof(y) ∈ {"string"} */
5     }
      else {
        break return ((deref x) - 1) /* x = ref y, tagof(y) ∈ {"number"} */
    } } }
  in (deref f)(200)
```

The flow analysis should compute that x = **ref** y at all program points, and that on lines 4 and 8, **tagof**$(y) \in \{$"number", "string"$\}$ and **tagof**$(y) \in \{$"number"$\}$, respectively. This information is enough to mechanically transform the program, replacing the `(deref x)` expressions with `tagcheck {"number","string"} (deref x)` on line 4 and `tagcheck {"string"} (deref x)` on line 8. Section 4.5 details a control-sensitive, heap-sensitive analysis that produces results such as this.

This analysis, like our type system, is mostly conventional. It is peculiar in populating the initial abstract heap with **tagof**$(y) \in \{$"number", "string"$\}$. A whole-program analysis might have used the application on line 9 to populate the heap with the argument value of `200`. In contrast, our analysis remains local but exploits the type annotation on `y`, thus determining that `tagof(y)` is in $\mathit{runtime}(\mathsf{Num} \cup \mathsf{Str}) = \{$"number", "string"$\}$.

We thus use types to modularize our flow analysis, so the analysis can remain strictly *intra*procedural. The time complexity of flow analysis is therefore a function of the size of individual functions in the program, which does not tend to grow as programs get larger. (Of course, the choice of function calls as modularity boundaries is not essential.) However, this does reduce precision, as we see below.

Assignment and Aliasing Our analysis is locally heap-sensitive and can type-check the following imperative variant of the example function:

```
let f = ref func(y) : Num ∪ Str → Num {
  let x = ref y in
  let _ = if (tagof (deref x) == "string") { setref x 1 }
```

```
          else { false } in
  (deref x) - 1 /* $x = \textbf{ref}\ y, \textbf{tagof}(y) \in \{"number"\}$ */ }
in (deref f)(200)
```

However, since we restart the analysis at function applications, we do not track non-local effects. In the following example, since foo(x) may assign either a number or a string to x, the analysis we present in section 4.5 simply restarts on all function applications. Thus we cannot insert a useful tagcheck around the subsequent **deref** x, so the example is untypable:

```
let g = ref func(y) : Num ∪ Str → Num {
  let x = ref y in
  let _ = setref x 10 in
  let _ = foo(x) in
  (deref x) /* $x = \textbf{ref}\ y, \textbf{tagof}(y) \in \{"number", "string"\}$ */}
in (deref g)("test")
```

More sophisticated analyses that tracked ownership or aliasing could make such examples typable.

Soundness Given that our flow analysis ignores actual arguments, is it sound? To show that a flow analysis is sound, we must define an acceptability relation and prove that statically computed abstract heaps remain acceptable under evaluation. However, here is a trivial variation of our example that violates acceptability:

```
let f = ref func(y) : Num ∪ Str → Num { /* ... as before ... */ }
in (deref f)(true)
```

The flow analysis ignores the actual argument **true** (tagged "boolean") and instead assumes that the type annotation is correct. That is, it assumes that at runtime, y is tagged either "number" or "string". Thus, we obtain only a weak preservation of solutions result (lemma 8).

Although flow analysis admits such mis-applied functions, the type system ensures that function applications are well-typed. Conversely, although the type system admits tagerrs at runtime, the flow analysis only inserts tagchecks that provably do not produce tagerrs. Hence, each component eliminates the other's weakness and in concert they combine to statically check programs that they cannot verify alone.

$$\begin{array}{lrcl}
\text{values} & V & = & x \mid c \mid l \mid \textsf{func}(x\cdots)\texttt{:}T\ \texttt{\{}\ M\ \texttt{\}} \mid \textsf{func}(x\cdots)\ \texttt{\{}\ M\ \texttt{\}} \\
\text{binding expressions} & B & = & V \mid \textbf{ref}\ V \mid \textbf{deref}\ V \mid \textsf{setref}\ V_1\ V_2 \mid op_n(V_1 \cdots V_n) \\
 & & \mid & \textsf{tagcheck}\ R\ V \mid \textsf{tagerr} \\
\text{unlabeled expressions} & N & = & \textbf{let}\ x\ \texttt{=}\ B\ \textsf{in}\ M \mid V_f(V\cdots) \\
 & & \mid & \textbf{if}\ (V)\ \texttt{\{}\ M_1\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ M_2\ \texttt{\}} \\
\text{labelled expressions} & M & = & N^{\hat{l}} \\
\text{stores} & S & = & \cdot \mid (l, V)\, S
\end{array}$$

Figure 4.8: Syntax of λ_S in CPS

4.5 Flow Analysis via CPS

A glaring issue with λ_S is that it has a single control operator, while real scripting languages support a plethora of control operators (section 2.1.4 and fig. 9.6). To avoid presenting an overly **break**-specific program analysis, we convert λ_S to CPS. CPS has the added advantage of naming intermediate terms, thereby simplifying our analysis. CPS is, however, not a requirement; we only use it for convenience.

4.5.1 CPS Transformation

Figure 4.8 specifies the syntax of CPS-λ_S, which, with the exception of V, is a syntactic restriction of λ_S. V includes administrative functions (explained shortly). We specify the CPS transformation using a technique developed by Sabry and Felleisen [75]. The transformation is defined by four mutually-recursive functions that respectively map programs, expressions, values, and evaluation contexts from direct-style to CPS:

$$\mathcal{P}_k : \sigma e \to SM \qquad \Phi : v \to V \qquad \mathcal{C}_k : e \to M \qquad \mathcal{K}_k : E \to V$$

For illustration, consider representative cases of these functions:

$$\begin{array}{rcl}
\mathcal{P}_k[\![(l, v) \cdots e]\!] & = & (l, \Phi(v)) \cdots \mathcal{C}_k[\![e]\!] \\
\Phi[\![\textsf{func}(x\cdots)\texttt{:}S\cdots \to T\ \texttt{\{}\ e\ \texttt{\}}]\!] & = & \textsf{func}(k,x\cdots)\texttt{:}(T \to \bot) \times S \cdots \to \bot\ \texttt{\{}\ \mathcal{C}_k[\![e]\!]\ \texttt{\}} \\
\mathcal{C}_k[\![E\langle v_f(v_{arg}\cdots)\rangle]\!] & = & \Phi[\![v_f]\!](\mathcal{K}_k[\![E]\!], \Phi[\![v_{arg}]\!]\cdots) \\
\mathcal{K}_k[\![E\langle \textbf{let}\ x\ \texttt{=}\ \bullet\ \textsf{in}\ e\rangle]\!] & = & \textsf{func}(x)\ \texttt{\{}\ \mathcal{C}_k[\![E\langle e\rangle]\!]\ \texttt{\}}
\end{array}$$

In the last case above, the transformation introduces functions not found in the source program to receive the bound value. Since all evaluation contexts are transformed into such "administrative"

$$\boxed{\begin{array}{ll} \widehat{S} : \hat{l} \to R & \text{abstract store} \\ \widehat{\Gamma} : x \to \widehat{V} & \text{abstract environments} \end{array}}$$

$$\widehat{V} \quad = \quad R \mid \mathsf{Ref}\ \hat{l} \mid \mathsf{Deref}\ \hat{l}\ R \mid \mathsf{LocTagof}\ \hat{l} \mid \mathsf{LocType}\ \hat{l}\ R$$

$$\frac{R_1 \subseteq R_2}{R_1 \sqsubseteq R_2}$$

$$\mathsf{LocTagof}\ \hat{l} \sqsubseteq \{\text{"string"}\}$$

$$\mathsf{LocType}\ \hat{l}\ R \sqsubseteq \{\text{"boolean"}\}$$

$$\mathsf{Deref}\ \hat{l}\ R \sqsubseteq R$$

$$\mathsf{Ref}\ \hat{l} \sqsubseteq \{\text{"location"}\}$$

Figure 4.9: Analysis Domains

functions, all control structures are thus transformed into applications of administrative functions.

For succinctness, we do not introduce continuation-passing operators, and instead let-bind operators' results. We elide the semantics of CPS-λ_S, since it is essentially the same as the semantics in fig. 4.3. This style of definition makes it easy to prove that direct-evaluation corresponds to CPS-evaluation, which is necessary to relate typing and flow analysis.

Lemma 7 (Soundness of CPS Transformation) *If $\sigma e \to \sigma' e'$ using reduction rule R, then $\mathcal{P}_k[\![\sigma e]\!] \twoheadrightarrow \mathcal{P}_k[\![\sigma' e']\!]$ using reduction rules R, E-Let, and $\widehat{\beta_v}$.*

In the lemma above, $\widehat{\beta_v}$ denotes the reduction rule for administrative functions (defined exactly as β_v). The lemma roughly states that intermediate redexes in CPS are applications of administrative functions and let-expressions.

4.5.2 Modular Flow Analysis

Figure 4.9 specifies our abstract values and the lattice that relates them. Abstract stores ($\widehat{S}$) map abstract locations ($\hat{l}$) to tag sets (R). (Abstract locations are labels on expressions, introduced by CPS.) On the other hand, abstract environments ($\widehat{\Gamma}$) map identifiers to abstract values ($\widehat{V}$) that will account for tag-tests.

For example, fig. 4.10 presents our example from the previous section in CPS. The comment on line 2 specifies the initial abstract environment, computed by applying *runtime* to the arguments.

```
let f = func(k, y):(Num → ⊥) × Num ∪ Str → ⊥ {
  // By V-Restart, k = {"function"}, y = {"number", "string"}
  let x = ref y in // By F-Alloc, x = Ref $\hat{l}$; $\hat{l}$ = {"number", "string"}
  let t1' = deref x in // By F-Deref, t' = Deref x $\widehat{S}(\hat{l})$
  let t1 = tagcheck {"number","string"} t1' in // By F-TagCheck, t1 = t1'
  let t2 = typeof t1' in // By F-Typeof, t2 = LocTypeof $\hat{l}$
  let t3 = (t2 === "string") in // By F-TypeIs-Str, t3 = LocType $\hat{l}$ {"string"}
    if (t3) { // By F-If-Split applied to $\hat{l}$
      k(0) } // By F-App, with $\hat{l}$ = {"number"}
    else {
      let t4' = deref x in // By F-Deref, t4' = Deref x $\widehat{S}(\hat{l})$; $\hat{l}$ = {"number"}
      let t4 = tagcheck {"number"} t4' in // By F-TagCheck, t4 = t4'
      let t5 = t4' - 1 in
        k(t5) } }
in let f' = deref f
in f'($k_{init}$,200)
```

Figure 4.10: tagcheck Insertion

The remaining comments specify how the abstract heap and environment are transformed by each statement. These transformation are *acceptable*, as specified by our acceptability relation (figs. 4.11 and 4.12).

Note that the user-written identifier x is bound to a heap-location. However, the CPS-introduced identifiers, which name the subexpressions that reason about x, are not heap-allocated. We exploit this stratification in our analysis domains to simplify the proof of soundness. The abstract heap and environment contain values that locally reason about the heap. For soundness, V-Restart therefore discards the abstract heap and uses *reset* and *del* to widen heap-dependent abstract values to simple tag sets.

Assignment and Aliasing In fig. 4.13, we account for the effects of assignments to tag sets. If a program sets an abstract location $\hat{l}$, then F-SetRef simply updates $\hat{l}$ in the abstract store of its continuation. However, the environment may bind identifiers to abstract values that reason about $\hat{l}$. Therefore, we use *del* to widen $\hat{l}$-dependent values to simple tag sets.

Local variables cannot reference each other. However, we use references to model mutable objects as well. A local variable bound to a mutable object is a reference to a reference, and these objects can be aliased. In these cases, we stop tracking the potentially-aliased abstract location, once again using *del*. F-Ref-Alias in fig. 4.13 tackles aliasing in **ref** expressions. Similar rules apply to other syntactic forms.

$$
\begin{array}{lcl}
del : \hat{l}, \widehat{\Gamma} \to \widehat{\Gamma} & & \\
del(\hat{l}, \cdot) & = & \cdot \\
del(\hat{l}, x : \textsf{Deref}\ \hat{l}\ R, \widehat{\Gamma}) & = & x : R, del(\hat{l}, \widehat{\Gamma}) \\
del(\hat{l}, x : \textsf{LocTagof}\ \hat{l}, \widehat{\Gamma}) & = & x : \{\textsf{"string"}\}, del(\hat{l}, \widehat{\Gamma}) \\
del(\hat{l}, x : \textsf{LocType}\ \hat{l}\ R, \widehat{\Gamma}) & = & x : \{\textsf{"boolean"}\}, del(\hat{l}, \widehat{\Gamma}) \\
del(\hat{l}, x : \textsf{Ref}\ \hat{l}, \widehat{\Gamma}) & = & x : r, del(\hat{l}, \widehat{\Gamma}) \\
del(\hat{l}, x : \widehat{V}, \widehat{\Gamma}) & = & x : \widehat{V}, del(\hat{l}, \widehat{\Gamma}) \\
reset(\widehat{\Gamma}) & = & del(\hat{l}_1, del(\hat{l}_2, ..., del(\hat{l}_n, \widehat{\Gamma}))), \forall \hat{l}_i \in \widehat{\Gamma}
\end{array}
$$

$\boxed{\widehat{\Gamma} \triangleright V \rightsquigarrow \widehat{V}}$

$$\frac{\cdot; x : runtime(T) \cdots, reset(\widehat{\Gamma}) \vDash M}{\widehat{\Gamma} \triangleright \textbf{func}(x \cdots)\texttt{:}T \cdots \to \bot\ \texttt{\{}\ M\ \texttt{\}} \rightsquigarrow \textsf{"function"}} \tag{V-Restart}$$

$$\widehat{\Gamma} \triangleright c \rightsquigarrow \delta_1(, c) \tag{V-Const}$$

$$\widehat{\Gamma} \triangleright x \rightsquigarrow \widehat{\Gamma}(x) \tag{V-Id}$$

$$\frac{\widehat{\Gamma} \triangleright V \rightsquigarrow \widehat{V} \qquad \widehat{V} \sqsubseteq \widehat{V'}}{\widehat{\Gamma} \triangleright V \rightsquigarrow \widehat{V'}} \tag{V-Sub}$$

Figure 4.11: Acceptability of Flow Analysis—Metafunctions and Values

Monotone Framework Our algorithm for computing tagchecks is a simple monotone framework [55] directly derived from the rules in figs. 4.11 and 4.12. The monotone framework computes the abstract store and environment at each labelled expression. We use this information to insert tagchecks into our programs.

Consider each expression of the form:

$\textbf{let}^{\hat{l}}\ r\ \texttt{=}\ \textbf{deref}\ x\ \texttt{in}\ M$

Let $\widehat{\Gamma}$ and $\widehat{S}$ be the computed abstract environment and store at $\hat{l}$. If $\widehat{\Gamma}(\hat{l}) = \textsf{Ref}\ \hat{l}'$, then we transform the expression to:

$\textbf{let}\ r'^{\hat{l}}\ \texttt{=}\ \textbf{deref}\ x\ \texttt{in}$
$\textbf{let}\ r\ \texttt{=}\ \texttt{tagcheck}\ \widehat{S}(\hat{l}')\ r'\ \texttt{in}$
M

For type-checking, this inserted tagcheck is mapped back to the original, direct-style program.

The administrative functions, if applied, can exponentially increase the size of programs. Therefore, we leave certain administrative redexes unapplied (e.g., continuations of **if**-expressions). The

$$\boxed{\widehat{S};\widehat{\Gamma} \vDash M}$$

$$\frac{\begin{array}{c}\widehat{\Gamma} \triangleright V \rightsquigarrow \widehat{V} \\ \widehat{S}; x : \widehat{V}, \widehat{\Gamma} \vDash M\end{array}}{\widehat{S};\widehat{\Gamma} \vDash \textbf{let}\ x\ \texttt{=}\ V\ \textsf{in}\ M} \qquad \text{(F-LetVal)}$$

$$\frac{\begin{array}{c}\widehat{\Gamma} \triangleright V \rightsquigarrow R \\ \hat{l} : R, \widehat{S}; x : \textsf{Ref}\ \hat{l}, \widehat{\Gamma} \vDash M\end{array}}{\widehat{S};\widehat{\Gamma} \vDash \textbf{let}^{\hat{l}} x\ \texttt{=}\ \textbf{ref}\ V\ \textsf{in}\ M} \qquad \text{(F-Alloc)}$$

$$\frac{\widehat{\Gamma} \triangleright V \rightsquigarrow \textsf{Ref}\ \hat{l} \qquad \widehat{S}(\hat{l}) = R \qquad \widehat{S}; x : \textsf{Deref}\ \hat{l}\ R, \widehat{\Gamma} \vDash M}{\widehat{S};\widehat{\Gamma} \vDash \textbf{let}\ x\ \texttt{=}\ \textbf{deref}\ V\ \textsf{in}\ M} \qquad \text{(F-Deref)}$$

$$\frac{\widehat{\Gamma} \triangleright V \rightsquigarrow \textsf{Deref}\ \hat{l}\ R \qquad \widehat{S}; x : \textsf{LocTagof}\ \hat{l}, \widehat{\Gamma} \vDash M}{\widehat{S};\widehat{\Gamma} \vDash \textbf{let}\ x\ \texttt{=}\ \textsf{tagof}\ V\ \textsf{in}\ M} \qquad \text{(F-Tagof)}$$

$$\frac{\widehat{\Gamma} \triangleright V \rightsquigarrow \textsf{LocTagof}\ \hat{l} \qquad \widehat{S}; x : \textsf{LocType}\ \hat{l}\ \{\text{"string"}\}, \widehat{\Gamma} \vDash M}{\widehat{S};\widehat{\Gamma} \vDash \textbf{let}\ x\ \texttt{=}\ V\ \texttt{===}\ \text{"string"}\textsf{in}\ M} \qquad \text{(F-TypeIs-Str}^{a}\text{)}$$

$$\frac{\widehat{\Gamma} \triangleright V \rightsquigarrow R \qquad \widehat{S}; x : R, \widehat{\Gamma} \vDash M}{\widehat{S};\widehat{\Gamma} \vDash \textbf{let}\ x\ \texttt{=}\ \textsf{tagcheck}\ R\ V\ \textsf{in}\ M} \qquad \text{(F-TagCheck)}$$

$$\frac{\widehat{\Gamma} \triangleright V \rightsquigarrow \textsf{LocType}\ \hat{l}\ R \qquad \widehat{S}[\hat{l} := R]; \widehat{\Gamma} \vDash M_1 \qquad \widehat{S}[\hat{l} := \widehat{S}(\hat{l}) \backslash R]; \widehat{\Gamma} \vDash M_2}{\widehat{S};\widehat{\Gamma} \vDash \textbf{if}\ \texttt{(}V\texttt{)}\ \texttt{\{}\ M_1\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ M_2\ \texttt{\}}} \qquad \text{(F-If-Split)}$$

$$\frac{\widehat{\Gamma} \triangleright V_f \rightsquigarrow \widehat{V_f} \qquad \widehat{\Gamma} \triangleright V \rightsquigarrow R \cdots}{\widehat{S};\widehat{\Gamma} \vDash V_f\texttt{(}V \cdots\texttt{)}} \qquad \text{(F-App)}$$

[a]F-TypeIsStr is easily generalized to arbitrary tags; we specialize it to strings for presentation only.

Figure 4.12: Acceptability of Flow Analysis (Essential Rules)

$$\frac{\widehat{\Gamma} \triangleright V_1 \rightsquigarrow \mathsf{Ref}\ \hat{l} \qquad \widehat{\Gamma} \triangleright V_2 \rightsquigarrow R \qquad \widehat{S}[\hat{l} := R]; x : \mathsf{Ref}\ \hat{l}, del(\hat{l}, \widehat{\Gamma}) \vDash M}{\widehat{S}; \widehat{\Gamma} \vDash \textbf{let}\ x\ \texttt{=}\ \mathsf{setref}\ V_1\ V_2\ \mathsf{in}\ M} \quad \text{(F-SetRef)}$$

$$\frac{\widehat{\Gamma} \triangleright V \rightsquigarrow \mathsf{Ref}\ \hat{l} \qquad \widehat{S}; x : r, del(\hat{l}, \widehat{\Gamma}) \vDash M}{\widehat{S}; \widehat{\Gamma} \vDash \textbf{let}\ x\ \texttt{=}\ \textbf{ref}\ V\ \mathsf{in}\ M} \quad \text{(F-Ref-Alias)}$$

Figure 4.13: Assignment and Aliasing

CPS transformation is therefore linear time and our flow analysis computes meets through administrative functions.

Complexity Our flow analysis is a monotonic ascent of a lattice of finite height. For a program of N terms our analysis computes an abstract store and environment at each term. The domain of abstract stores and environments are both of size $O(N)$. The range of the abstract store is R, and $|R|$ is a constant. The range of the abstract environment is $\widehat{V}$, where $\widehat{V}$ contains the elements of R. The additional elements of $\widehat{V}$ are incomparable with each other and are all less than the elements of R. Hence, the height of $\widehat{V}$ is just 1 greater than the height of R (i.e., $O(1)$). Thus, the analysis needs time quadratic in the program size. In practice, our prototype implementation type-checks real-world JavaScript programs in seconds on modest machines. This is the payoff of using types to modularize the program analysis at function boundaries.

In contrast, a program analysis of untyped, higher-order code, such as 0CFA [78], must compute the set of all functions that may be applied ($O(N)$ functions) at all call sites ($O(N)$ call sites). The range of abstract environments in 0CFA thus has size $O(N)$, which makes the algorithm cubic.

Soundness In addition to fig. 4.12 and fig. 4.13, we require trivial rules for cases where our flow analysis cannot determine useful information. These additional rules admit all other expressions, except tagerrs and possibly-faulty tagchecks. Soundness also requires auxiliary rules that reason about the concrete values in the store that are introduced by evaluation. We elided the concrete store from figures 4.12 and 4.13 for clarity; in the following lemmas, we introduce it.

Lemma 8 (Soundness) *If $\hat{S}, \cdot \vDash SM$ and $SM \rightarrow S'M'$ then either:*

i. $\hat{S}', \cdot \vDash S'M'$, or

ii. M is a β_v-redex of shape $(\textbf{func}(x \cdots)\ :\ T \cdots \rightarrow \bot.N)\ (V \cdots)$, where for some V, $\delta_1(\textit{tagof}, V) \notin runtime(T)$.

4.5.3 Combining Typing and Flow Analysis

We can now prove a stronger progress result that eliminates tagerrs.

Theorem 2 (Strengthened Progress) *If:*

i. $\Sigma; \cdot \vdash e : T$,

ii. $\Sigma \vdash \sigma$, *and*

iii. $\widehat{S}; \cdot \models \mathcal{P}_k[\![\sigma e]\!]$,

then either:

i. $e \in v$, *or*

ii. There exist σ' *and* e'*, such that* $\sigma e \to \sigma' e'$.

Proof: This follows from lemma 6, with the possibility of tagerrs eliminated by inspection of fig. 4.12—flow analysis does not admit expressions with tagerrs. ■

Theorem 2 requires a corresponding, combined preservation theorem.

Theorem 3 (Combined Preservation) *If:*

i. $\Sigma; \cdot \vdash e : T$,

ii. $\Sigma \vdash \sigma$,

iii. $\widehat{S}; \cdot \models \mathcal{P}_k[\![\sigma e]\!]$, *and*

iv. $\sigma e \to \sigma' e'$,

then there exist Σ' *and* $\widehat{S}'$*, such that:*

i. $\Sigma'; \cdot \vdash e' : T$,

ii. $\Sigma' \vdash \sigma'$,

iii. $\Sigma \subseteq \Sigma'$, *and*

iv. $\hat{S}'; \cdot \models \mathcal{P}_k[\![\sigma' e']\!]$.

Proof: Conclusions (i.), (ii.), and (iii.) follow immediately from lemma 5. For conclusion (iv.), apply lemma 7 to hypothesis (iv.) to get a reduction sequence, $\mathcal{P}_k[\![\sigma e]\!] \twoheadrightarrow \mathcal{P}_k[\![\sigma' e']\!]$. Apply lemma 8 at each step, eliminating case (ii.) of the lemma as follows. By lemma 7, intermediate expressions are not β_v-redexes, so case (ii.) does not apply. Suppose e itself has an active β_v-redex:

$$e = E\langle \textsf{func}\texttt{(}x\cdots\texttt{)}\ \texttt{:}\ U\cdots \rightarrow S\ \texttt{\{}\ e_f\ \texttt{\}(}v\cdots\texttt{)}\rangle$$

Once transformed to CPS, e has the form

$$\textsf{func}\texttt{(}k\texttt{,}x\cdots\texttt{)}\ \texttt{:}\ (S \rightarrow\bot) \times U \cdots \rightarrow\bot\ \texttt{\{}\ M_p\ \texttt{\}(}V\cdots\texttt{)}$$

where $V\cdots$ are $v\cdots$ in CPS. Since e is typed, there exists a Γ such that:

$$\Sigma;\Gamma \vdash \textsf{func}\texttt{(}x\cdots\texttt{)}\ \texttt{:}\ U\cdots \rightarrow S\ \texttt{\{}\ e_f\ \texttt{\}(}v\cdots\texttt{)} : S$$

For all v, $\Sigma;\Gamma \vdash v : U$ by inversion. Hence $\delta_1(\textsf{tagof}, v) \in \mathit{runtime}(U)$. Since conversion to CPS does not change tags, $\delta_1(\textsf{tagof}, v) = \delta_1(\textsf{tagof}, V)$, case (ii.) of lemma 8 does not apply. ■

4.6 Related Work

Typed Scheme Typed Scheme [86, 87] is a type system designed to admit Scheme idioms. Typed Scheme uses *occurrence typing* to account for type tests and type predicates. However, occurrence typing is unsound in the presence of imperative features; thus, it is "turned off" when imperative features are used. Unlike the Scheme programs that Typed Scheme types, programs in mainstream scripting languages make heavy use of imperative features, which we handle.

Technically, we develop a type system and flow analysis that are complementary by design (Lemmas 6 and 8), which combine soundly (Theorems 2 and 3), and which can be enriched independently within the framework of these two lemmas. We conjecture that a similar structure could be extracted from Typed Scheme, as the type system is augmented with meta-functions that update the environment (see Typed Scheme's use of $\Gamma+$ and $\Gamma-$ to affect the environment, and *combpred* to prop type tests to the context in **if** (fig. 4.14)). We believe these are similar to transfer functions for dataflow analyses. However, Typed Scheme is not organized in this manner.

$$\frac{\Gamma \vdash e_1 : \tau_1; \phi_1 \qquad \Gamma + \phi_1 \vdash e_2 : \tau_2; \phi_2 \qquad \Gamma - \phi_1 \vdash e_3 : \tau_3; \phi_3 \\ \vdash \tau_2 <: \tau \qquad \vdash \tau_3 <: \tau \qquad \phi = \mathit{combpred}(\phi_1, \phi_2, \phi_3)}{\Gamma \vdash (\mathbf{if}\, e_1\, e_2\, e_3) : \tau; \phi}$$

Figure 4.14: If-splitting in Typed Scheme [86]

Intensional Polymorphism Intensional polymorphism [17] provides a `typecase` construct that allows programs to inspect and dispatch on the type of values at runtime. This requires a term-level representation of types at runtime, which is only possible when the static and dynamic semantics of a programming language are co-designed. The present work, Typed Scheme, and other retrofitted type systems (discussed below) do not have access to their types at runtime. Type dispatch in a retrofitted type system happens indirectly. For example, Typed Scheme uses predicates [86], while our work relies on the relationship between static types and runtime tags (section 4.3).

Other Retrofitted Type Systems Soft Scheme [91] performs type inference for Scheme programs. It handles the full language of the time, and has a limited form of if-splitting. It does not pay any additional attention to the interaction of types and control flow. This is reasonable because it, like Typed Scheme, is focused on Scheme programs that are mostly functional. However, this means that it too cannot handle the kinds of examples shown in this chapter and found in many scripting languages.

Anderson et al. [5] tackle type inference for JavaScript. However, their language is extremely limited, and their type system cannot tackle the idioms discussed in this chapter (section 4.1).

Heidegger and Thiemann's [43] *recency types* account for ad hoc object initialization patterns that are pervasive in JavaScript, but does not address the problems that this chapter does. Our work does not account for objects. Our preliminary investigation suggests that the two approaches are complementary and can fruitfully be combined.

Henglein and Rehof [45] present a translation of Scheme to ML that uses type inference to minimize runtime projections. However, their "type system does not model control flow information" [45, Section 6.5], which is the goal of our work.

Diamondback Ruby [33] is a type system and type inference for Ruby. Although its type language includes union types, it does not account for type-tests to discriminate members of unions, which is the focus of our work. The authors state that "support for occurrence types would be useful future work".

Types and Flow Analysis Shivers shows how control-flow can be extended to account for type-tests [78, Chapter 9]. However, whole-program analysis for functional and object-oriented languages is non-modular and expensive [31] or difficult to make effective [30]. Meunier et al. [63] develops a modular analysis for an untyped language by using contracts as sources and sinks for abstract values. We exploit type annotations in the same manner. Since all functions have type annotations, our flow analysis problem is significantly more tractable than in an untyped language with optional contracts.

Jensen et al. [51, 52] and MrSpidey [31] use flow analysis to recover precise type-like information for arbitrary JavaScript and Scheme programs, respectively. A significant advantage of flow analysis is that it does not require type annotations. Our work requires and exploits type annotations to achieve modularity, which leads to quadratic time complexity in theory that appears to translate into practice (section 4.5.2).

There are known equivalences between various type systems and control-flow analyses, e.g., Heintze [44], Nielson and Nielson [68], and Palsberg and O'Keefe [69]. The aforementioned works extend type systems to calculate information that is conventionally calculated by flow analyses. In contrast, our type system is oblivious to control flow information (fig. 4.5). We use a separate flow analysis to account for control-sensitive and heap-sensitive reasoning (section 4.4). We independently prove typing and flow analysis sound, then show that they combine in a simple way (section 4.5.3).

Definite assignment analysis is a commonly used flow analysis that augments typing (e.g., see the Java Language Specification [36, Chapter 16]). Definite assignment analysis conservatively ensures that variables are assigned before they are used. Hence, the analysis rejects programs as untypable when all variables are not definitely assigned. In contrast, our analysis augments the type system to accept programs that would otherwise be untypable.

Chapter 5

Typing Objects

In most statically-typed object-oriented languages, an object's type or class enumerates its member names and their types. This set of names is finite and the names are first-order, barring cumbersome reflection APIs. In contrast, in JavaScript, reflection is trivial, class hierarchies are fluid, objects' shapes are amorphous, and member names are first-class strings that can be computed dynamically. These features are difficult to type-check.

This chapter[1] presents *fluid object types*, a type language for describing the dynamic, reflective idioms employed by JavaScript. Fluid object types have two novel features: they employ *string patterns* to describe possibly-infinite collections of fields and *presence annotations* to precisely state the position of inherited fields. We demonstrate that fluid object types account for many programming patterns that are conventionally untypable. Furthermore, chapter 8 demonstrates that fluid object types provide a simple account of language-based Web sandboxes.

5.1 λ_S^{ob}: A Core Calculus of Lightweight Objects

λ_{JS} (chapter 2) is fully capable of modeling JavaScript's objects. However, λ_{JS} also models various other details of JavaScript that are inessential for understanding objects. We thus present fluid object types using a smaller core calculus of lightweight objects.

Figure 5.1 defines the syntax and semantics of λ_S^{ob}, a calculus of lightweight objects. The objects of λ_S^{ob} are extensible records with inheritance and field-deletion. λ_S^{ob} distinguishes "parent" as

[1]This chapter is based on joint work with Joe Gibbs Politz [39].

$$
\begin{array}{rcll}
P & = & \cdots & \text{Patterns} \\
c & = & \mathit{num} \mid \mathit{str} \mid \mathit{bool} & \text{Constants} \\
v & = & c \mid \textbf{func}\texttt{(}x : T\texttt{)}\texttt{.}e \mid \Lambda\alpha <: T.e \mid \texttt{\{ } \mathit{str}\texttt{:}v\cdots \texttt{ \}} & \text{Values} \\
e & = & x \mid v \mid e\texttt{(}e\texttt{)} \mid \texttt{\{ } \mathit{str}\texttt{: } e\cdots \texttt{ \}} \mid e\texttt{[}e\texttt{]} \mid e\texttt{[}e \texttt{ = } e\texttt{]} \mid \textbf{delete } e\texttt{[}e\texttt{]} \mid e\texttt{(}T\texttt{)} & \text{Expressions} \\
 & \mid & \textbf{if } \texttt{(}e_1\texttt{)} \ e_2 \ \textbf{else} \ e_3 \mid e_1 \ \textbf{hasfield} \ e_2 \mid e \ \textbf{matches} \ P & \\
 & \mid & \textbf{fix } \texttt{(}f\texttt{:}T\texttt{)}\texttt{.}e \mid e_1 \texttt{ + } e_2 & \\
E & = & \bullet \mid E\texttt{(}e\texttt{)} \mid v\texttt{(}E\texttt{)} \mid \texttt{\{ } \mathit{str}\texttt{: } v\cdots \ \mathit{str}\texttt{:}E\texttt{, } \mathit{str}\texttt{:}e\cdots \texttt{ \}} \mid E\texttt{[}e\texttt{]} \mid v\texttt{[}E\texttt{]} & \text{Contexts} \\
 & \mid & v\texttt{[}v \texttt{ = } E\texttt{]} \mid E\texttt{[}e \texttt{ = } e\texttt{]} \mid v\texttt{[}E \texttt{ = } e\texttt{]} \mid \textbf{delete } E\texttt{[}e\texttt{]} \mid \textbf{delete } v\texttt{[}E\texttt{]} & \\
 & \mid & \textbf{if } \texttt{(}E\texttt{)} \ e_2 \ \textbf{else} \ e_3 \mid E \ \textbf{hasfield} \ e \mid v \ \textbf{hasfield} \ E \mid E \ \textbf{matches} \ P & \\
 & \mid & E \texttt{ + } e \mid v \texttt{ + } E &
\end{array}
$$

$$
\begin{array}{ll}
\beta_v & \texttt{(}\textbf{func}\texttt{(}x : T\texttt{)}\texttt{.}e\texttt{)(}v\texttt{)} \hookrightarrow e[x/v] \\
\text{E-Fix} & \textbf{fix}\texttt{(}f\texttt{:}T\texttt{)}\texttt{.}e \hookrightarrow e[f/\textbf{fix}\texttt{(}f\texttt{:}T\texttt{)}\texttt{.}e] \\
\text{E-TApp} & (\Lambda\alpha <: S.e)\texttt{(}T\texttt{)} \hookrightarrow e[\alpha/T] \\
\text{E-GetField} & \texttt{\{ } \cdots \mathit{str}\texttt{: } v\cdots \texttt{ \}[}\mathit{str}\texttt{]} \hookrightarrow v \\
\text{E-Inherit} & \texttt{\{ } \mathit{str} \texttt{ : } v\cdots \texttt{"parent": } v_p \texttt{ \}[}\mathit{str}_x\texttt{]} \hookrightarrow v_p\texttt{[}\mathit{str}_x\texttt{]}, \text{ if } \mathit{str}_x \notin (\mathit{str}\cdots) \\
\text{E-Update} & \texttt{\{ } \cdots \mathit{str}\texttt{: } v\cdots \texttt{ \}[}\mathit{str} \texttt{ = } v_x\texttt{]} \hookrightarrow \texttt{\{ } \cdots \mathit{str}\texttt{: } v_x \cdots \texttt{ \}} \\
\text{E-Create} & \texttt{\{ } \mathit{str}\texttt{: } v\cdots \texttt{ \} [}\mathit{str}_x \texttt{ = } v_x\texttt{]} \hookrightarrow \texttt{\{ } \mathit{str}_x\texttt{: } v_x\texttt{, } \mathit{str}\texttt{: } v\cdots \texttt{ \}}\text{if } \mathit{str}_x \notin (\mathit{str}\cdots) \\
\text{E-Del} & \textbf{delete } \texttt{\{ } \cdots \ \mathit{str}_x\texttt{: } v_x \ \cdots \texttt{ \} [}\mathit{str}_x\texttt{]} \hookrightarrow \texttt{\{ } \cdots \texttt{ \}} \\
\text{E-NoDel} & \textbf{delete } \texttt{\{ } \mathit{str}\texttt{: } v \ \cdots \texttt{ \} [}\mathit{str}_x\texttt{]} \hookrightarrow \texttt{\{ } \mathit{str}\texttt{: } v \ \cdots \texttt{ \}}\text{if } \mathit{str}_x \notin (\mathit{str}\cdots) \\
\text{E-IfTrue} & \textbf{if } \texttt{(}\textbf{true}\texttt{)} \ e_2 \ \textbf{else} \ e_3 \hookrightarrow e_2 \\
\text{E-IfFalse} & \textbf{if } \texttt{(}\textbf{false}\texttt{)} \ e_2 \ \textbf{else} \ e_3 \hookrightarrow e_3 \\
\text{E-Has} & \texttt{\{ } \cdots \mathit{str}\texttt{:}v\cdots \texttt{ \}} \ \textbf{hasfield} \ \mathit{str} \hookrightarrow \textbf{true} \\
\text{E-HasNot} & \texttt{\{ } \mathit{str}\texttt{:}v\cdots \texttt{ \}} \ \textbf{hasfield} \ \mathit{str} \ \hookrightarrow \textbf{false}, \text{ when } \mathit{str}' \notin (\mathit{str}\cdots) \\
\text{E-Match} & \mathit{str} \ \textbf{matches} \ P \hookrightarrow \textbf{true}, \ \mathit{str} \in P \\
\text{E-}\neg\text{Match} & \mathit{str} \ \textbf{matches} \ P \hookrightarrow \textbf{false}, \ \mathit{str} \notin P \\
\text{E-String+} & \mathit{str}_1 + \mathit{str}_2 \hookrightarrow \mathit{str}_1 \mathit{str}_2
\end{array}
$$

$$\text{E-Cxt} \quad E\langle e_1\rangle \to E\langle e_2\rangle, \text{ when } e_1 \hookrightarrow e_2$$

$$
\begin{array}{rcl}
\textbf{let } x \texttt{ = } e_1 \textbf{ in } e_2 & \equiv & \texttt{(}\textbf{func}\texttt{(}x\texttt{:}T\texttt{)}\texttt{.}e_2\texttt{)(}e_1\texttt{)} \\
\textbf{let rec } x \texttt{ = } e_1 \textbf{ in } e_2 & \equiv & \texttt{(}\textbf{func}\texttt{(}x\texttt{:}T\texttt{)}\texttt{.}e_2\texttt{)(}\textbf{fix } x\texttt{:}T\texttt{.}e_1\texttt{)} \\
\textbf{func}\texttt{(}x_1\texttt{:}T_1\cdots x_n\texttt{:}T_n\texttt{)}\texttt{.}e & \equiv & \textbf{func}\texttt{(}x_1\texttt{:}T_1\texttt{)}\texttt{.}\cdots\texttt{.}\textbf{func}\texttt{(}x_n\texttt{:}T_n\texttt{)}\texttt{.}e \\
e_f\texttt{(}e_1\cdots e_n\texttt{)} & \equiv & e_f\texttt{(}e_1\texttt{)}\cdots\texttt{(}e_n\texttt{)}
\end{array}
$$

Figure 5.1: Syntax and Semantics of λ_S^{ob}

a reference to the parent object for inheritance (E-Inherit). However, "parent" is otherwise undistinguished; a program can retrieve, update, or even delete "parent". Field names are first-class strings, not first-order labels. A single program may work with an arbitrary collection of field names, or even dynamically construct names by concatenating strings (E-String+). λ_S^{ob} programs can use reflection (**hasfield**) to determine which fields are present on an object. In addition, λ_S^{ob} programs can use **matches** to determine if a string matches a particular pattern. We leave the representation of patterns unspecified. (Other languages use a mix of regular expressions and ad hoc testing to achieve the same effect.) Finally, λ_S^{ob} includes explicit type annotations and instantiations; we introduce types in section 5.3.

The primary distinction between λ_S^{ob} and λ_{JS} is that λ_S^{ob} lacks mutable state for clarity of presentation. As discussed in section 5.5.1, all our proofs and theorems are over an imperative λ_S^{ob}. λ_S^{ob} does have additional expressions for string-concatenation and string-matching, but these are trivially expressible as primitives in λ_{JS}(section 2.1.6).

5.2 Idiomatic λ_S^{ob}: Type-Checking Challenges

The examples below demonstrate that λ_S^{ob} can model many characteristic uses of objects in scripting languages. (In appendix B, we translate these examples to various scripting languages.) For now, we leave the static types unspecified. We type-check these examples in section 5.3.5, after developing fluid object types.

Example 1: Prototype-based Objects λ_S^{ob} can easily encode prototype inheritance:

```
let Rect = { "area": func(self:?) . self["x"] * self["y"] } in
let Cuboid = { "parent": Rect,
               "vol": func(self) . self["area"](self) * self["z"] } in
let shape = { "x": 2, "y": 5, "z": 10: "parent": Cuboid } in
let vol = shape["vol"](shape) // vol is 100
```

Above, methods are simply function valued-fields that take an explicit `self` argument. With this encoding, a program can directly access the `"parent"` field to redefine methods or apply methods to other objects.

Example 2: Extracting Methods Supporting method-extraction promotes code reuse across structurally similar values. For example, arrays may have a collection of utility methods in their common parent:

```
let ArrParent = { "slice": func(self:?,begin:?,end:?). ··· , ··· } in
let arr1 = { "0": 3, "1": 20, "2": 59, "length": 3, "parent": ArrParent }
```

The implementation of `slice` may only require that `self["len"]` be defined; it does not actually need `self` to be an array. Therefore, this single slice method can be copied and applied to other kinds of collections. This exact scenario occurs frequently in JavaScript programming. Web browsers have other kinds of collections, such as HTMLNodeList, that are not arrays and thus do not have various array methods:

```
let nodeList = { "0": htmlElementA, "1": htmlElementB, "2": htmlElementC,
                 "len": 3, "parent": HTMLNodeListParent } in
let eltArray = ArrParent["slice"](nodeList,0,1)
// returns an array containing htmlElementA and htmlElementB
```

However, in the code above, slice is applied to an HTMLNodeList to convert it to an array and make other utility methods (e.g., map and reduce) available.

Example 3: Classes λ_S^{ob} can also encode classes by having methods close over a particular value of self. Although explicitly passing self to each method is enormously flexible, it is error-prone.[2] We build on the prototypal shapes of Example 1 below:

```
let rec shape2 = {
  "x": 2, "y": 5, "z": 10",
  "_class_": Cuboid,
  "parent": {
    "vol": func() . shape2["_class_"]["vol"](shape2)
  }
} in
let f = shape2["vol"]
let vol2 = f() // vol2 is still 100, f closes over shape2
```

In this encoding, methods do not require an explicit self argument, but the underlying methods are still accessible via the "_class_" field.[3] This encoding allows methods themselves to manipulate objects' structure. However, it also allows other code to freely modify objects. We will use types to hide the _class_ field from other code.

Example 4: Ad Hoc Private Fields All fields are public in λ_S^{ob}'s lightweight objects. In such object systems, it is common to use a convention, such as "field names that begin with an underscore are private".[4] This convention is easily violated by malicious or buggy code. A module may wish to protect some of the fields in objects in its implementation, and only provide client code with access to a safe lookup function.

[2]JavaScript suffers exactly this error; `this` is an implicit argument but is supplied at each method call (fig. 2.5).

[3]Python and Ruby have similar encodings; methods close over their self argument, but the underlying method is still accessible.

[4]Python and Dart employ such conventions.

For example, a simple dynamic check to ensure that untrusted code does not access fields that begin and end with underscores is as follows:

```
let safeGetField = Λα <: ?.func(obj:?,fieldName:?,default:?).
  if (fieldName matches "_.*_") default
  else if (obj hasfield fieldName) obj[fieldName] else default in
safeGetField(?)({ "_private_": 42, "pub": 23 },
                "_private_", 0) // returns 0
```

These checks are notoriously difficult to implement in full-fledged scripting languages. Our type system for λ_S^{ob} will demonstrate techniques for statically verifying that such code is correct.

Example 5: Dictionaries as Dictionaries There is no need to implement dictionaries in λ_S^{ob}; objects can be used as dictionaries themselves. Some care must be taken in implementation, however, as a naïve approach may accidentally extract a method from the object's parent, or the parent itself:

```
let ObjectParent = { "serialize": func(self:?).···, ··· } in
let dict = { "habitat": "a natural home or environment",
             "park": "a large enclosed piece of ground",
             "parent": ObjectParent } in
dict["habitat"] // returns "a natural home or environment"
dict["serialize"] // returns func(self) ... !
dict["parent"] // returns ObjectParent!
```

To avoid such mishaps, programmers should guard dynamic dictionary accesses:[5]

```
let safeAssign = Λα <: ?. Λβ <:?. func(dict:?,word:Str,value:?).
                  dict["w_" + word = value]
let safeLookup = Λα <: ?.func(dict:?,word:Str,default:?).
  let lookup = "w_" + word in
  if (dict hasfield lookup) dict[lookup]
  else default
```

This example presents a safe alternative, which prefixes words with "w_" and then uses a dynamic check to ensure only words are accessed. Our type system reasons about field tests to give reasonable static types to dictionaries.

Type System Features These examples elicit a baseline feature set to typecheck:

[5]A fact the developers of Google Docs know all too well. At the time of this writing, typing `__proto__`, JavaScript's parent, crashes Google Docs: `www.google.com/support/forum/p/Google+Docs/thread?tid=0cd4a00bd4aef9e4`.

$$\begin{array}{rcll} L & = & P \mid \alpha \mid L_1 \cap L_2 \mid L_1 \cup L_2 \mid L_1 L_2 \mid \overline{L} & \text{String patterns} \\ b & = & \textsf{Num} \mid \textsf{Bool} & \text{Base types} \\ \alpha & = & \cdots & \text{Type variables} \\ T & = & b \mid T_1 \to T_2 \mid \mu\alpha.T \mid \top \mid \forall \alpha <: S.T & \\ & \mid & L \mid \{L_1^{p_1} : T_1 \cdots L_n^{p_n} : T_n, L_A : \textbf{abs}\} & \text{Types} \\ p & = & \circ \mid \downarrow \mid \uparrow & \text{Field presence} \\ \Gamma & = & \cdot \mid \Gamma, x : T \mid \Gamma, \alpha <: T & \text{Environments} \end{array}$$

$\boxed{\Gamma \vdash T}$

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

$$\frac{\alpha \in dom(\Gamma)}{\Gamma \vdash \alpha} \quad \text{(WF-TVar)}$$

Figure 5.2: Types for λ_S^{ob}

1. Addition and deletion of fields,
2. Prototype-based inheritance with method extraction,
3. Class-like inheritance with bound and unbound method extraction,
4. Invariants on pattern-based field conventions, and
5. Field-presence guards on field access.

5.3 Types for Objects

The preceding examples guide the development of fluid object types in this section. Our full type language is in fig. 5.2, but we motivate and incrementally develop fluid object types in this section. In subsequent sections, we detail the associated type system.

5.3.1 Simple Records

Structural record types are a natural starting point, since λ_S^{ob} does not have classes. Recall that record types are finite maps from field names to field types:

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

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

Figure 5.3: Functions and Predicates over String Types

Record types can type trivial programs, but not those in section 5.2.

5.3.2 Field Patterns

Record types are too simple because they statically enumerate all their fields' names. Interesting programs use computed field lookups, where the exact field name is not statically known. Consider the type of `c` in the following example:

```
let getCoord = func(pt:{"x" : Num,"y" : Num}, c:?) . pt[c]
```

The intended type for `c` is not an arbitrary string (Str), but a string in the set {"x", "y"}. We thus extend our types with precise string types, L-types, that denote sets of strings:

$$\begin{array}{lcll} L & = & P & \text{String patterns} \\ T & = & \cdots \mid L & \end{array}$$

The exact representation of L-types is not significant; we only require that the functions and predicates in fig. 5.3 be defined.

Type-checking field lookup with L-typed strings is straightforward. If `c` has type {"x", "y"}, then `pt[c]` has type Num. In general, if *fld* has type L and *obj* has type $\{str_1 : T_1 \cdots str_n : T_n\}$ then at runtime *obj*[*fld*] may lookup any $str_i \in L$. Thus, the type of *obj*[*fld*] is the join of the matching fields' types.

Patterns Replace Field Names L-types allow computed field names, but record types still specify a finite list of fields. They cannot express the type of the dictionary in Example 5, so we reuse L-types to generalize field names to patterns:

$$T = \cdots \mid \{L_1 : T_1 \cdots L_n : T_n\}$$

This is the first of the two key features of fluid object types. We can now describe an object with an arbitrary collection of fields. For example, in the following type, all fields that begin with an underscore have type Num and all other fields have type Bool.[6]

$$\{\_.* : \mathsf{Num}, \overline{\_.*} : \mathsf{Bool}\}$$

Something odd happened here—this type seems to describe an object with an infinite number of fields. Since objects are finite dictionaries, we could interpret this type as a specification of fields' types *if they are present*; if a field is absent then indexing that field gets stuck, but typing is preserved. We can do better and address this problem with *presence annotations* in the next section.

In summary, fluid object types use patterns (L-types) to describe collections of fields in objects. There are three common classes of L-types:

- The L-type for the set of all strings is the usual Str-type.
- An L-type that represents a singleton set, $\{str\}$, is the type of the string literal str. In $e_1\texttt{[}e_2\texttt{]}$, if $e_2 : \{str\}$ then the expression is a conventional, known-label field lookup.
- An L-type that represents a possibly-infinite set of strings is useful for typing operations on computed field names. For example, in object update, $e_1\texttt{[}e_2\texttt{ = }e_3\texttt{]}$, if the type of e_2 is the co-finite set $\overline{\{\text{"parent"}\}}$, then the operation does not affect "parent" and does not affect the inheritance chain.

5.3.3 Presence Annotations

Conventional structural object types do not expose the position of members on the inheritance chain; types are "flattened" to include inherited members. However, the "parent" field of λ_S^{ob} objects allows programs to distinguish inherited fields. If we flatten object types, all such programs would be untypable.

Fluid object types allow us to expose the precise structure of the inheritance chain with ease. In the following type, move is present on the parent:

$$\{\texttt{"parent"} : \{\texttt{"move"} : \mathsf{Num} \rightarrow \mathsf{Num}\}\}$$

[6]We often use regular expressions to describe patterns. However, regular expressions are not fundamental; any decision procedure over strings is adequate.

This precision unfortunately makes conventional uses of class hierarchies untypable. Functions that consume subtypes of a particular class are agnostic to the position of methods on the inheritance chain; all that matters are methods' types. However, the type above requires move to be present on precisely the first parent, and does not admit objects that inherit but do not override move.

To remedy this, we introduce the second feature of fluid object types. We add *presence annotations* to fields and a pattern for fields that are *definitely absent*:

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

We interpret presence annotations as follows:

- $L^{\downarrow} : T$ indicates that all fields $str \in L$ are *definitely present on the object itself* with type T. These fields are not inherited and are not absent.
- $L^{\uparrow} : T$ indicates that all fields $str \in L$ are either *present on the object itself or along the inheritance chain* with type T.
- $L^{\circ} : T$ indicates that all fields $str \in L$ *may be present on the object itself.* If a field str is present it has type T, but it may be absent.
- The $L_A : \mathbf{abs}$ annotation indicates that all fields $str \in L_A$ are *definitely absent* on the object; however, they may be present higher up on the inheritance chain. Looking up a definitely absent field therefore does not fail if the field is inherited.

Notation We use two abbreviations to simplify the syntax of fluid objects.

- We elide writing an empty set of absent fields, $\emptyset : \mathbf{abs}$. For example:

$$\{\text{"x"}^{\downarrow} : \mathsf{Num}, \text{"y"}^{\downarrow} : \mathsf{Num}\} = \{\text{"x"}^{\downarrow} : \mathsf{Num}, \text{"y"}^{\downarrow} : \mathsf{Num}, \emptyset : \mathbf{abs}\}$$

- It is often convenient for the set of absent fields to be the complement of all other fields; we abbreviate this to $\star : \mathbf{abs}$. For example, $\star$ represents $\overline{\{\text{"x"}, \text{"y"}, \text{"parent"}\}}$ in the following type:

$$\{\text{"x"}^{\downarrow} : \mathsf{Num}, \text{"y"}^{\downarrow} : \mathsf{Num}, \text{"parent"}^{\downarrow} : P, \star : \mathbf{abs}\}$$

Flexibility and Guarantees of Presence Annotations The $L^{\downarrow} : T$ and $L^{\uparrow} : T$ annotations ensure that indexing an L-typed field produces a T-typed value; programs cannot get stuck indexing fields. They also allow other fields with the same name but different types to exist further up the inheritance chain, but these do not affect the type of object indexing.

The $L^{\circ} : T$ annotation provides weaker guarantees. For example, if obj has the type:

$$\{\texttt{"x"}^{\circ} : \mathsf{Num}, \texttt{"parent"}^{\downarrow} : \{\texttt{"x"}^{\downarrow} : \mathsf{Bool}\}\}$$

then obj["x"] may return either field. For full generality, we present a typing rule that determines that obj["x"] has type $\mathsf{Num} \sqcup \mathsf{Bool}$. What would be more useful is an operator that narrows the $\circ$ to a $\downarrow$.

5.3.4 Reflection

Consider typing Example 5 from section 5.2. We might give dict the following type:

$$\{\mathtt{w\_.*}^{\circ} : \mathsf{Str}\}$$

With this type, the expression dict["w_habitat"] is not typable. The type indicates that "w_habitat" may not exist on the object, so we must also compute the type of dict["parent"]["w_habitat"]. However, "parent" is not specified.

We must first establish that dict["w_habitat"] is definitely present. In Example 5, the safeLookup function guards dictionary indexing with a **hasfield** check:

```
if (dict hasfield lookup) dict[lookup] else default
```

The type system can *if-split* [86] to account for such guards and narrow the types of dict and lookup in the true branch. We present a simple if-splitting rule in section 5.5; here we describe the types in the true-branch after narrowing.

In the true-branch, we have established that the value of lookup names a member of dict; we can thus narrow a possibly absent member ($\circ$) to definitely present ($\downarrow$). However, we do not know exactly which member to narrow. In particular, the following narrowing is wrong:

$$\mathsf{dict} : \{\mathtt{w\_.*}^{\downarrow} : \mathsf{Str}\}$$

The type above states that all members are definitely present. However, the program only establishes that a single string is definitely present.

We express this by splitting the pattern `w_.*` into two components, a type variable α that represents the string bound to `lookup` and $\mathtt{w\_.*} \cap \overline{\alpha}$, the remainder of the pattern. Thus in the true-branch, the type-environment, Γ', modifies the enclosing type environment, Γ, as follows:

$$\begin{aligned} \Gamma' &= \Gamma, \alpha <: P, \mathsf{word} : \alpha, \mathsf{dict} : \{\alpha^{\downarrow} : \mathsf{Str}, P \cap \overline{\alpha}^{\circ} : \mathsf{Str}\} \\ & \quad \text{where } P = \mathtt{w\_.*} \end{aligned}$$

The patterns used above include type variables and set operators:

$$L = P \mid \alpha \mid L_1 \cap L_2 \mid L_1 \cup L_2 \mid L_1 L_2 \mid \overline{L}$$

The final specification of fluid object types and string pattern types is in fig. 5.2. With these types, we can type-check the examples from section 5.2.

5.3.5 Type-Checking Examples

With fluid object types introduced, we now revisit the examples from section 5.3.5.

Example 1 The types for Rect and Cuboid are as follows:

$$\mathsf{Rect} : \quad \left\{ \text{"area"}^{\downarrow} : \{\text{"x"}^{\downarrow} : \mathsf{Num}, \text{"y"}^{\downarrow} : \mathsf{Num}\} \to \mathsf{Num} \right\}$$

$$\mathsf{Cuboid} : \left\{ \begin{array}{l} \text{"vol"}^{\downarrow} : \left\{ \begin{array}{l} \text{"x"}^{\downarrow} : \mathsf{Num}, \text{"y"}^{\downarrow} : \mathsf{Num}, \text{"z"}^{\downarrow} : \mathsf{Num}, \\ \text{"area"}^{\uparrow} : \{\text{"x"}^{\downarrow} : \mathsf{Num}, \text{"y"}^{\downarrow} : \mathsf{Num}\} \to \mathsf{Num} \end{array} \right\} \to \mathsf{Num}, \\ \text{"parent"}^{\downarrow} : \mathsf{Rect} \end{array} \right\}$$

We abuse notation slightly for brevity; in Cuboid we use Rect as an abbreviation for the whole type. The presence annotations on these types are interesting. On Rect's "area" method, the argument omits listing absent fields and only specifies fields the method needs. This allows an object like shape, which has a "z" field, to be freely used with "area". Cuboid["vol"] marks area as inherited ($\uparrow$), since it is agnostic to its position on the inheritance chain.

Example 2 $\mathsf{ArrParent}$ requires $\mathsf{"slice"}$'s argument to have a Num-typed $\mathsf{"len"}$ field and consistently-typed numeric fields:

$$\forall \alpha <: \top . \mu \beta . \left\{ \begin{array}{lcl} \mathsf{"slice"}^{\downarrow} & : & \{\mathsf{Dec}^{\circ} : \alpha, \mathsf{"len"}^{\downarrow} : \mathsf{Num}\} \\ & \rightarrow & \{\mathsf{Dec}^{\circ} : \alpha, \mathsf{"len"}^{\downarrow} : \mathsf{Num}, \mathsf{"parent"}^{\downarrow} : \beta\} \\ \ldots & & \ldots \end{array} \right\}$$

$$\mathsf{Dec} = \mathtt{0\,|\,[1\text{-}9]\,[0\text{-}9]*}$$

Fields that match Dec may be present; if they are, they must have type Num as indicated by the regular expression pattern Dec. If $\mathsf{ArrParent}$ is instantiated with the type $\mathsf{HTMLElement}$, then $\mathsf{"slice"}$ can be freely used on the object in Example 2.

Example 3 The types of Rect and Cuboid are the same as in Example 1 in this class encoding, but note that $\mathsf{shape2["vol"]}$ is closed over its self-argument. We can give $\mathsf{shape2}$ the following type, hiding $\mathsf{\_class\_}$, to prevent external code from observing its internals:

$$\{\mathsf{"x"}^{\downarrow} : \mathsf{Num}, \mathsf{"y"}^{\downarrow} : \mathsf{Num}, \mathsf{"z"}^{\downarrow} : \mathsf{Num}, \mathsf{"vol"}^{\uparrow} : \rightarrow \mathsf{Num}\}$$

Example 4 The type of $\mathsf{safeGetField}$ is:

$$\forall \alpha <: \top . \{\overline{\mathsf{\_.*\_}}^{\circ} : \alpha\} \rightarrow \mathsf{Str} \rightarrow \alpha \rightarrow \alpha$$

The pattern $\overline{\mathsf{\_.*\_}}^{\circ} : \alpha$ indicates that all non-underscored fields have type α, if they are present.

Example 5 The types in this example are:

$$\begin{array}{lcl} \mathsf{ObjectParent} & : & \{\mathsf{"serialize"} : \top \rightarrow \mathsf{Str}\} \\ \mathsf{dict} & : & \forall \alpha <: \top . \{\mathsf{w\_.*} : \alpha, \mathsf{"parent"}^{\downarrow} : \mathsf{ObjectParent}, \mathsf{"serialize"} : \mathbf{abs}\} \\ \mathsf{safeAssign} & : & \forall \alpha <: \top . \forall \beta <: \{\mathsf{w\_.*}^{\circ} : \alpha\} . \beta \rightarrow \mathsf{Str} \rightarrow \alpha \rightarrow \beta \\ \mathsf{safeLookup} & : & \forall \alpha <: \top . \{\mathsf{w\_.*}^{\circ} : \alpha\} \rightarrow \mathsf{Str} \rightarrow \alpha \rightarrow \alpha \end{array}$$

We assume that $\mathsf{serialize}$ can serialize arbitrary values. Notably, dict's type allows $\mathsf{serialize}$ to be called, but the types of $\mathsf{safeAssign}$ and $\mathsf{safeLookup}$ ensure that they cannot access $\mathsf{serialize}$ and only manipulate

$\boxed{\Gamma \vdash S <: T}$

$$\frac{\Gamma \vdash T_a <: S_a \qquad \Gamma \vdash S_r <: T_r}{\Gamma \vdash S_a \to S_r <: T_a \to T_r}$$

$$\frac{}{\Gamma \vdash b <: b}$$

$$\Gamma \vdash T <: \top$$

$$\frac{\Gamma \vdash S <: T[\alpha/\mu\alpha.T]}{\Gamma \vdash S <: \mu\alpha.T}$$

$$\frac{\Gamma \vdash S[\alpha/\mu\alpha.S] <: T}{\Gamma \vdash \mu\alpha.S <: T}$$

$$\Gamma \vdash \alpha <: \alpha$$

$$\frac{\alpha <: S \in \Gamma \qquad \Gamma \vdash S <: T}{\Gamma \vdash \alpha <: T}$$

$$\frac{\Gamma, \alpha <: U \vdash S <: T}{\Gamma \vdash (\forall \alpha <: U.S) <: (\forall \alpha <: U.T)}$$

$$\frac{L_1 \subseteq L_2}{\Gamma \vdash L_1 <: L_2} \tag{S-Str}$$

$$\frac{\begin{array}{c}(1)\forall i,j.\text{if } L_i \cap M_j \neq \emptyset \text{ then } p_i <: q_j \text{ and } \Gamma \vdash S_i <: T_j \\ (2)\bigcup_i^{1\cdots m} M_i \subseteq \bigcup_j^{1\cdots n} L_j \cup L_A \qquad (2')M_A \subseteq L_A \\ (3)\forall j.\text{if } M_j \cap L_A \neq \emptyset \text{ then } q_j = \circ \text{ or } q_j = \uparrow \\ (4)\forall j.\text{if } q_j = \uparrow \text{ then } \Gamma \vdash \mathit{inherit}_\Gamma(\{L_1^{p_1} : S_1, \cdots, L_n^{p_n} : S_n, L_A : \mathbf{abs}\}, M_j) <: T_j\end{array}}{\Gamma \vdash \{L_1^{p_1} : S_1, \cdots, L_n^{p_n} : S_n, L_A : \mathbf{abs}\} <: \{M_1^{q_1} : T_1, \cdots, M_m^{q_m} : T_m, M_A : \mathbf{abs}\}} \tag{S-Ob}$$

$\boxed{p <: q}$

(p-Refl)	(p-Maybe)	(p-Inherit)	(p-IMaybe)
$p <: p$	$\downarrow <: \circ$	$\downarrow <: \uparrow$	$\downarrow <: \circ$

Figure 5.4: Algorithmic Subtyping

words in the dictionary.

$\boxed{\mathit{inherit}_\Gamma : S \times L \to T}$

$$\mathit{inherit}_\Gamma(\{L_1^{p_1} : T_1 \cdots L_n^{p_n} : T_n, L_A : \mathbf{abs}\}, L_Q) = \bigsqcup \{T_i \mid L_Q \cap L_i \neq \emptyset\} \cup \mathcal{T}_P$$

$$\mathcal{T}_P = \begin{cases} \emptyset & \text{if } L_Q \subseteq \bigcup\{L_i \mid p_i \neq \circ\} \text{ and} \\ & \neg\exists L_k.\texttt{"parent"} \in L_k \\ \{\mathit{inherit}_\Gamma\,(T_k, L_Q \cap (L_A \cup \bigcup\{L_i | p_i = \circ\}))\} & \text{if } L_Q \subseteq \bigcup_i^{1\cdots n} L_i \cup L_A \text{ and} \\ & \exists L_k.\texttt{"parent"} \in L_k \end{cases}$$

Figure 5.5: The *inherit* Metafunction

5.4 Subtyping

This section presents algorithmic subtyping for λ_S^{ob}. Figure 5.4 is the entire algorithmic subtyping relation. We present fluid object types along with equirecursive μ-types and bounded quantification (kernel rule). Some form of recursive type is necessary to type-check objects—we choose equirecursive μ-types. Bounded quantification is commonly used to encode data structures; we also employ bounded quantification in our account of reflection. The majority of the algorithmic subtyping relation is conventional. The two interesting rules are S-Str and S-Ob.

Subtyping string types with S-Str uses pattern inclusion. For example, if patterns are defined as regular languages, inclusion is decidable. When patterns include variables, these inclusion constraints can be discharged by existing string solvers [47]. Discharging these patterns may require constraints on L-bounded type variables in Γ. The appropriate Γ is always unambiguous from context, therefore we write $L_1 \subseteq L_2$ instead of $\Gamma \vdash L_1 \subseteq L_2$.

Algorithmic subtyping for objects, S-Ob, is a generalization of algorithmic subtyping of records. Recall that algorithmic subtyping for records combines the declarative width, depth, and permutation subtyping rules. S-Ob combines generalizations of depth, width, and permutation subtyping, in addition to a *flattening* rule for inherited fields. For object types, $\Gamma \vdash S <: T$ if and only if they satisfy the four antecedents of S-Ob:

1. (Permutation and Depth) The types of fields with overlapping names must be subtypes. In addition, $p <: q$ is a partial order on presence-annotations. Intuitively, $p <: q$ means that T can "forget" that a field is definitely present.

2. (Width) The field patterns of S must include the field patterns of T. Therefore, T can "hide" fields of S.

3. (Depth) Any field that is absent (**abs**) in S may be possibly absent ($\circ$) or inherited ($\uparrow$) in T. This is depth subtyping for absent fields; T can "forget" that a field is absent and introduce it with $\circ$ or $\uparrow$ annotations.

4. (Flattening) For an inherited field ($\uparrow$) to appear on T, subtyping must ensure that the field is defined, with the appropriate type, somewhere on the inheritance chain of S. The metafunction *inherit* calculates this type on S and the pattern of the $\uparrow$-annotated field.

Flattening When $\Gamma \vdash S <: T$, the subtype S can describe the exact position of fields in the inheritance chain, e.g.

$$S = \{\text{"x"}^{\downarrow} : \mathsf{Num}, \text{"parent"}^{\downarrow} : \{\text{"y"}^{\downarrow} : \mathsf{Str}\}, \text{"y"} : \mathbf{abs}\}$$

However, T may lose this information and flatten the type to

$$T = \{\text{"x"}^{\downarrow} : \mathsf{Num}, \text{"y"}^{\downarrow} : \mathsf{Str}\}$$

The purpose of *inherit* (fig. 5.5) is to ensure that $\uparrow$-annotated fields are appropriately flattened in the supertype, T. For an object type T, $\mathit{inherit}_{\Gamma}(T, L_Q)$ calculates the join of all field on S with patterns that may intersect L_Q, given constraints on L-bounded type variables in Γ. This includes all of the fields on the inheritance chain of S. Recall that fluid object types allow fields with the same pattern to have different types at different points in the inheritance chain. Consider the following type:

$$T = \{\text{"x"}^{\downarrow} : \mathsf{Num}, \text{"z"}^{\downarrow} : \mathsf{Num}, \text{"parent"}^{\downarrow} : \{\text{"z}^{\downarrow} : \mathsf{Bool}\}\}$$

With this definition of T, $\mathit{inherit}(T, \text{"z"}) = \mathsf{Num}$,[7] since indexing a T-typed object with "z" cannot produce the Bool-valued field in the parent. However, consider the slightly different type:

$$T' = \{\text{"x"}^{\downarrow} : \mathsf{Num}, \text{"z"}^{\circ} : \mathsf{Num}, \text{"parent"}^{\downarrow} : \{\text{"z}^{\downarrow} : \mathsf{Bool}\}\}$$

[7]An omitted Γ argument to *inherit* denotes an empty type environment.

In T', since "z" may be absent ($\circ$) indexing may produce either "z" field. Therefore, *inherit*$(T', \text{"z"}) = \mathsf{Num} \sqcup \mathsf{Bool}$. Finally, consider the following type:

$$U = \{\text{"z"}^\circ : \mathsf{Num}\}$$

In U, "z" may be absent, but U does not specify the type of "parent". It is thus unknown if "parent" exists, much less if it has a "z" field and what its type might be. Therefore, *inherit*$(U, \text{"z"})$ is undefined. We can see this by looking at the definition of *inherit*, specifically in defining $\mathcal{T}_P$ for U and "z". The first case of $\mathcal{T}_P$ does not apply, because the annotation on "z" is $\circ$, contradicting the side condition. The second case does not apply either, because there is no "parent" field on U.

inherit requires a join operator, $\sqcup$, over types. The join operator must satisfy:

$$(S \sqcup T = U) \Rightarrow (S <: U \wedge T <: U)$$

We elide the full definition; the interesting case is for object types:

$$\{L_i^{p_i} : S_i, \cdots, L_A : \mathbf{abs}\} \sqcup \{M_j^{q_j} : T_j, \cdots, M_A : \mathbf{abs}\}$$

Which is a pairwise intersections of patterns:

$$\left\{ \begin{array}{l} (L_i \cap M_j)^{p_i \sqcup q_j} : S_i \sqcup T_j \cdots, \\ (L_i \cap M_A)^\circ : S_i \cdots, (L_A \cap M_j)^\circ : T_j \cdots, \\ L_A \cap M_A : \mathbf{abs} \end{array} \right\}$$

For computing the join of the presence annotations $p_i \sqcup q_j$, we use the lattice induced by the $p <: q$ relation in fig. 5.4.[8]

Lemma 9 (Decidability of Subtyping) *If the functions and predicates on patterns (fig. 5.3) are decidable, then the subtype relation is finite-state.*

Proof: By coinduction on the subtyping judgments.

[8]The join operator, as presented, may introduce a number of $\emptyset$ field patterns. Inspection of the typing rules shows that empty patterns are inconsequential.

$$ty(num) = \mathsf{Num} \qquad ty(bool) = \mathsf{Bool}$$

$\boxed{\Gamma \vdash e : T}$

$$\frac{\Gamma \vdash e : S \qquad \Gamma \vdash S <: T}{\Gamma \vdash e : T}$$

$$\frac{\Gamma(x) = T}{\Gamma \vdash x : T}$$

$$\Gamma \vdash c : ty(c)$$

$$\frac{\Gamma, x : S \vdash e : T}{\Gamma \vdash \textbf{func } (x : S).e : S \rightarrow T}$$

$$\frac{\Gamma \vdash e_f : S \rightarrow T \qquad \Gamma \vdash e_a : S}{\Gamma \vdash e_f(e_a) : T}$$

$$\frac{\Gamma, \alpha <: S \vdash e : T}{\Gamma \vdash \Lambda\alpha <: S.e : \forall\alpha <: S.T}$$

$$\frac{\Gamma \vdash e : \forall\alpha <: U.T \qquad \Gamma \vdash S <: U}{\Gamma \vdash e\langle S\rangle : T[\alpha/S]}$$

$$\frac{\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Gamma \vdash e_2, e_3 : T}{\Gamma \vdash \textbf{if } (e_1)\ e_2 \textbf{ else } e_3 : T}$$

Figure 5.6: Typing Basics

5.5 Typing

Figure 5.6 presents the basic elements of the typing relation, which includes a conventional account of functions and bounded quantification. The interesting typing rules are in fig. 5.7 and discussed below.

T-Str is the typing rule for string literals, which ascribes a string *str* the singleton L-type $\{str\}$. T-Str+ concatenates two string patterns.

T-Object is the typing rule for object literals. In an object literal, all fields are definitely present ($str^{\downarrow}$) and all field names are statically known. Thus, each field name is a singleton L-type. In addition, since the object has no more fields, the complement of its fields is definitely absent ($\star : \textbf{abs}$).

$$\Gamma \vdash \mathit{str} : \{\mathit{str}\} \qquad \text{(T-Str)}$$

$$\frac{\Gamma \vdash e_1 : L_1 \qquad \Gamma \vdash e_2 : L_2}{\Gamma \vdash e_1 + e_2 : L_1 L_2} \qquad \text{(T-Str+)}$$

$$\frac{\Gamma \vdash e_1 : S_1 \cdots \Gamma \vdash e_n : S_n}{\Gamma \vdash \texttt{\{ } \mathit{str}_1 \texttt{:}\ e_1, \cdots, \mathit{str}_n \texttt{:}\ e_n \texttt{ \}} : \{\mathit{str}_1^{\downarrow} : S_1, \cdots, \mathit{str}_n^{\downarrow} : S_n, \star : \mathbf{abs}\}} \qquad \text{(T-Object)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_o : T \qquad \Gamma \vdash e_f : L \qquad \Gamma \vdash e_v : S \qquad L \subseteq \bigcup\{L_1 \cdots L_n\} \\ T = \{L_1^{p_1} : T_1, \cdots, L_n^{p_n} : T_n, L_A : \mathbf{abs}\} \qquad \forall L_i \cap L \neq \emptyset . \Gamma \vdash S <: T_i\end{array}}{\Gamma \vdash e_o \texttt{[} e_f \texttt{ = } e_v \texttt{]} : T} \qquad \text{(T-Update)}$$

$$\frac{\begin{array}{c}\Gamma \vdash e_o : T \qquad \Gamma \vdash e_f : L \qquad L \subseteq \bigcup\{L_1 \cdots L_n\} \qquad \forall L \cap L_i \neq \emptyset . p_i = \circ \\ T = \{L_1^{p_1} : T_1, \cdots, L_n^{p_n} : T_n, L_A : \mathbf{abs}\}\end{array}}{\Gamma \vdash \textsf{delete}\ e_o \texttt{[} e_f \texttt{]} : T} \qquad \text{(T-Delete)}$$

$$\frac{\begin{array}{c}U = \{L_1^{p_1} : S_1, \cdots, L_n^{p_n} : S_n, L_A : \mathbf{abs}\} \qquad \Gamma \vdash e_o : U \qquad \Gamma \vdash e_f : L_Q \\ T = \mathit{inherit}_{\Gamma}(U, L_Q)\end{array}}{\Gamma \vdash e_o \texttt{[} e_f \texttt{]} : T} \qquad \text{(T-GetField)}$$

$$\frac{\begin{array}{c}\Gamma(o) = \{\cdots L^{\circ} : S \cdots\} \qquad \Gamma(f) = L \qquad \Gamma \vdash e_3 : T \\ \Gamma, \alpha <: L, f : \alpha, o : \{\cdots \alpha^{\downarrow} : S, (L \cap \overline{\alpha})^{\circ} : S \cdots\} \vdash e_2 : T\end{array}}{\Gamma \vdash \textsf{if}\ \texttt{(}o\ \textsf{hasfield}\ f\texttt{)}\ e_2\ \textsf{else}\ e_3 : T} \qquad \text{(T-HasField-Split)}$$

Figure 5.7: Typing Fluid Objects

Typing Object Operations Field update and field deletion are not affected by inheritance, so typing these operations does not treat "parent" specially. Both T-Update and T-Delete allow the field name to be an arbitrary L-typed expression, and not just a string constant. Therefore, at runtime, the index expression may reduce to any string, $\mathit{str} \in L$. The necessary restrictions are:

- For T-Update, if a field $(L_i^{p_i} : T_i)$ overlaps at all with the type of the field name to update (L), then the type of the value (S), must be a subtype of the type of the field (T_i).
- For T-Delete, if a field overlaps with the type of the field name to delete, then it must be possibly absent ($\circ$-annotated).

Object types are thus invariant under updates and deletion; we do not account for strong updates in this dissertation, but they are supported by other type systems for scripting languages (section 5.7).

T-GetField types field lookup and must therefore account for inheritance. To do so, it uses the *inherit* metafunction from object subtyping (section 5.4).

Typing Reflection T-HasField-Split is a simple *if-splitting* rule that accounts for the use of reflection in the conditional, and refines the types of o and f in the true-branch when the check succeeds. There are various sophisticated if-splitting techniques for typing complex conditionals and control [16, 87] including flow typing, which is presented in this dissertation (chapter 4). T-HasField-Split could be adapted to the aforementioned systems. [9]

T-HasField-Split introduces a new type variable α in the true branch, bounded by $\alpha <: L$, as the type of the string f. Using α, it splits the type of pattern L, marking α as definitely present ($\downarrow$) whereas $L \cap \overline{\alpha}$ remains possibly absent ($\circ$). Therefore, an `o[f]` expression in the true branch has exactly the type S.

5.5.1 Soundness and Mutable State

We prove standard progress and preservation theorems for λ_S^{ob}. The proofs are over an extended semantics with mutable references, reference types, and store typings. The presentation of fluid object types in this chapter omits references for clarity, but we account for state in our proofs and prototype implementation.

Theorem 4 (Preservation) *If $\Sigma \vdash \sigma$, $\cdot \vdash e : T$, and $\sigma e \rightarrow \sigma' e'$, then there exists a Σ', such that $\Sigma \subseteq \Sigma'$, $\Sigma' \vdash \sigma'$, and $\Sigma'; \cdot \vdash e' : T$.*

Theorem 5 (Progress) *$\Sigma \vdash \sigma$ and $\cdot \vdash e : T$ then either $e \in v$ or there exist σ' and e' such that $\sigma e \rightarrow \sigma' e'$.*

5.6 Implementation

Our type-checker for JavaScript uses fluid object types to type-check objects. The core language of the type-checker is λ_S, which is closely related to λ_S^{ob}, which is presented in this chapter. Therefore, the implementation is essentially a direct encoding of the material in this chapter.

Pattern Representations We use two representations for string patterns: finite sets of strings and finite automata. For finite automata, we use the representation and decision procedure of Hooimeijer and Weimer [48]. Their implementation is fast and based on mechanically proven principles. A thin wrapper transparently converts patterns represented as sets to equivalent finite automata when necessary. The representation of patterns is thus fully abstract to the type-checker.

[9] This single typing rule is adequate for type-checking, but the proof of preservation requires auxiliary rules in the style of Typed Scheme [86].

Type-Checking Experiments We type-check the λ_S^{ob} examples in this chapter, various other λ_S^{ob} benchmarks, and $7,000$ lines of JavaScript code. The JavaScript code we type-check consist of two suites of programs:

- 11 Google Chrome Experiments and Google Gadgets (chapter 7), and
- The ADsafe runtime and various sample ADsafe widgets (chapter 8).

We make two observations about typing objects in JavaScript.

- JavaScript programs heavily manipulate the DOM, which is a massive, object-oriented API with functions and objects for manipulating HTML, CSS, local storage, the canvas, etc. These APIs are specified in various IDL files, [10] which are a type-like interface definitions. We process $4,000$ lines of IDL to build a type environment of object types for Web applications.
- In chapter 8, we present a type-based approach for language-based Web sandboxes. Fluid object types are instrumental for this verification; they are needed to describe the interfaces of Web sandboxes.

Performance Subtyping patterns (fig. 4.4) and type well-formedness (fig. 5.2) suggest two possible performance bottlenecks: both require pairwise pattern intersection checks and subtyping requires pattern inclusion checks, which are reducible to pattern intersection. Finite automata intersection takes exponential time and it is easy to construct synthetic examples that demonstrate the worst-case time complexity of subtyping. Fortunately, we have reason to believe that realistic programs work with a small collection of "interesting" patterns; most patterns are constant strings. The only interesting pattern in the JavaScript standard library is for array indices. ADsafe and Google Caja, discussed above, have two and eight patterns respectively. We test our implementation with patterns from the aforementioned systems. On these patterns, our type-checker is fast; it runs various benchmarks in approximately one second on an Intel Core i5 processor.

5.7 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.

[10] `www.w3.org/TR/WebIDL`

Semantics of Scripting Languages There are various semantics for scripting languages (chapter 2 and [34, 58, 79]) that model each language in detail. This chapter focuses on type-checking a core calculus of objects and elides many features and details of individual scripting languages. Our semantics also abstracts the plethora of string-matching operations available in real scripting languages into a single pattern matching construct, and makes object-reflection manifest for type-checking.

Extensible Records The representation of objects in λ_S^{ob} is derived from extensible records, surveyed by Fisher and Mitchell [28] and Bruce, et al. [13]. Wand presents a type inference algorithm for objects encoded as records [89]. Unlike these languages, field names in λ_S^{ob} are first-class strings; λ_S^{ob} includes operators to enumerate over fields and test for the presence of fields, since these are typical of scripting languages. Our fluid object types account for these features using presence-annotations and field-name patterns that are related to types for scripting languages, discussed below.

Types and Contracts for Untyped Languages There are various type systems retrofitted onto untyped languages. We discuss those that support objects.

Strongtalk [12] is a typed dialect of Smalltalk that uses *protocols* to describe objects. Field 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, fluid object types are purely structural, though we do not anticipate any difficulty incorporating brands.

Our work shares features with various JavaScript type systems. In the type system of Anderson, et al. [5], objects' fields may be potentially present; it employs strong updates to turn these into definitely present fields. Recency types [43] support field type-changes during initialization. Zhao's type system [95] also allows unrestricted object extension, but omits prototypes. In contrast to these works, our fluid object types do not support strong updates. We instead allow possible-absent fields to turn into definitely-present fields via reflection, which they do not support. Strong updates would be fruitful to type initialization patterns. In these type systems, field names are first-order labels. Thiemann's [83] type system for JavaScript allows first-class strings as field names, which we generalize to field patterns. In addition, we allow inheritance chains to be precisely typed by distinguishing possibly-inherited fields from fields that are immediately present. These are useful for typing features of scripting languages (section 5.2).

RPython [4] 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 give guarantees without staging restrictions.

DRuby [33] does not account for reflection in general. However, as a special case, An, et al. [3] 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 reflection directly.

System D [16] uses dependent refinements to type dynamic dictionaries. Fluid object types can type-check dictionaries as a special case of objects; we also account for inheritance, recursive objects and imperative state. System D accounts for richer control-dependent type reasoning than the single if-splitting rule presented in this chapter. However, chapter 4 accounts for control and and state based type reasoning using other techniques. The authors of System D suggest integrating a string decision procedure to reason about dictionary keys. We use DPRLE [48] to support exactly this style of reasoning.

Heidegger, et al. [42] present *dynamically*-checked contracts for JavaScript that use regular expressions to describe objects. Our implementation uses regular expressions for *static* checking.

Regular Expression Types Regular tree types and regular expressions can describe the structure of XML documents (e.g., XDuce [49]) and strings (e.g., XPerl [81]). 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' field names. Our string patterns thus allow individual objects to have semi-determinate shapes. Like XPerl, field names are simply strings, but our strings are used to index objects, which are not modeled by XPerl.

Chapter 6

Assisted Type Refactoring

In the next chapter, we evaluate our JavaScript type-checker on a body of third-part code. However, manually type-refactoring programs is labor-intensive. Therefore, this chapter presents a type-refactoring tool that uses dynamic analysis to infer many annotations[1]. We employed this tool produce the results in chapter 7.

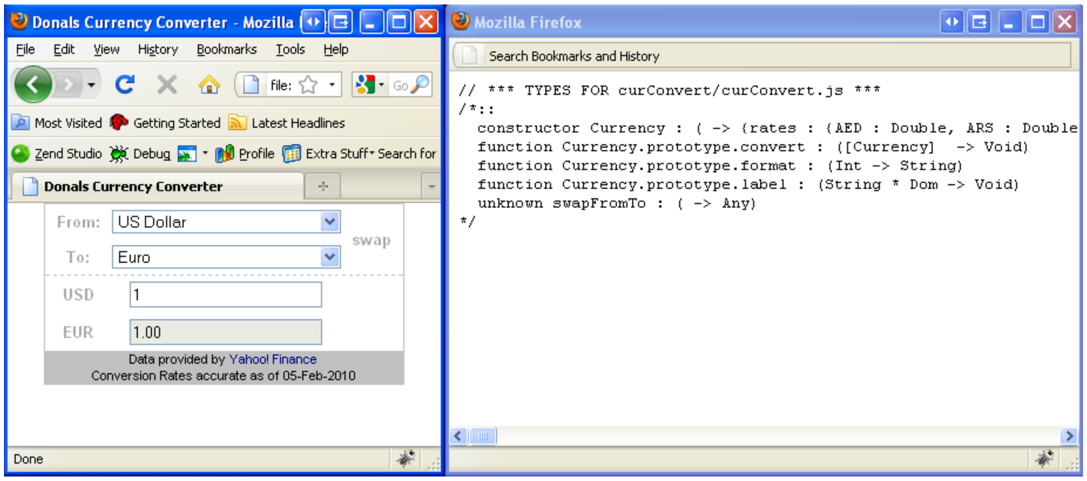
6.1 Approximating Types by Runtime Instrumentation

Refactoring a large, untyped program to use types is too costly to do at once. Therefore, there are various mechanisms that enable typed and untyped code to interoperate in a single program [29, 85, 92]. We adapt Typed Racket's mechanisms for interoperability [85]. Since JavaScript does not have modules, we encapsulate typed code in a closure, effectively creating programs with just two modules.

However, merely enabling typed and untyped code to interoperate does not help programmers refactor their code for types. To inject type-annotations into untyped code in any programming language, programmers have to do the following:

- They have to identify the program fragment that they want to make type-safe.
- They have to add type annotations to the fragment, and they have to change code to appease the type-checker.

[1]This chapter is based on joint work with Claudiu Saftoiu [76].


Donals Currency Converter - Mozilla
File Edit View History Bookmarks Tools Help
Most Visited
Getting Started
Latest Headlines
Zend Studio
Debug
Profile
Extra Stuff
Search for
Donals Currency Converter
From: US Dollar
To: Euro
swap
USD 1
EUR 1.00
Data provided by Yahoo! Finance
Conversion Rates accurate as of 05-Feb-2010
Done
Mozilla Firefox
Search Bookmarks and History

```
// *** TYPES FOR curConvert/curConvert.js ***
/*::
  constructor Currency : ( -> {rates : {AED : Double, ARS : Double
  function Currency.prototype.convert : ([Currency]  -> Void)
  function Currency.prototype.format : (Int -> String)
  function Currency.prototype.label : (String * Dom -> Void)
  unknown swapFromTo : ( -> Any)
*/
```

Figure 6.1: JSTrace output on Firefox

- Finally, they have to write type annotations for all the untyped free identifiers in the fragment.

These tasks can be daunting. They are particularly difficult when the programmer doing the type-refactoring is not the original author. They're harder still when the program is poorly documented. Because we evaluate our work on various third-party programs found on the Web, we encountered these problems ourselves.

Type inference is a possible solution. For Standard ML, inference is particularly convenient because inference computes principal types. However, principal types aren't necessary and inference need not even be sound, so long as we type-check after inference.

We have built a tool that infers (possibly unsound) type annotations by runtime instrumentation [67]. From the programmer's perspective, the tool displays a window with currently computed type annotations. As the program runs and control-paths are exercised, the annotations "grow". At any point, the programmer can conclude that the program has been exercised enough and stop execution. Our tools insert the computed types into the original source. The programmer can then try to type-check the program. Even when the computed annotations are insufficient, they are never entirely wrong, since they're based on values observed at runtime. They help a maintenance programmer get started on the task of type-refactoring.

The instrumentation builds an abstract heap, mapping labeled functions to abstract values. The instrumentation records sets of abstract arguments and results of these labeled functions. Constants are abstracted to their runtime tags, objects are abstracted by abstracting their fields, and functions are abstracted to their labels. We map the abstract heap to a collection of types in a natural way. Runtime tags map to types, abstract objects map to object types, multiple abstract values in a single set map to a union type, and function labels l map to the type annotation of the function l.[2]

In chapter 7, we use this tool to automatically generate type annotations for third-party JavaScript programs. Many of these programs are undocumented, and the generated type annotations make type-refactoring much easier.

[2]We signal an error on cycles, though self-application does not occur in practice.

Chapter 7

Evaluation I: Documentation

This and the next chapter evaluate our JavaScript type-checker on a body of third-party code. In this chapter,[1] we present our experience type-refactoring 15 small programs that were authored by different programmers. We demonstrate that some of this code is awful and undocumented. Once typed, the type annotations become a form of provably correct documentation for these programs. This chapter thus demostrates that our types are an effective way to document JavaScript code.

7.1 Type-Checking Gadgets and Chrome Experiments

We evaluate our work by fully type-refactoring an assortment of Chrome Experiments and Google Desktop Gadgets.[2] These programs are written by several authors and employ many different programming styles. Many are a number of years old and have little or no documentation. Type-refactoring these programs demonstrates that our type-checker admits several programming styles. Note that these are all small programs that one could simply rewrite in a new typed programming language. When programs grows by a few orders of magnitude, rewriting becomes harder and type-refactoring becomes more relevant. Due to a lack of resources, we have not tried to type-refactor larger JavaScript programs.

Figure 7.1 lists these programs and information that we discuss below. In all cases, we first ran our inference tool (section 6.1) to generate annotations. The columns labeled "Type Annotations" state the number of annotations *correctly inserted* by the tool described in chapter 6 (Auto) and the

[1]This chapter is based on joint work with Claudiu Saftoiu [76].
[2]`www.chromeexperiments.com` and `desktop.google.com`

		Refactorings		Annotations	
Program	LOC	Fixes	Problems	Auto	Manual
analogclock	112	0	6	13	0
animation	70	0	0	4	1
burncanvas	157	10	8	11	1
catchit	165	7	9	6	3
countdown	129	2	12	4	0
hashapass	257	1	7	13	7
light	151	8	19	3	7
metronome	106	1	4	10	2
morse	275	8	5	12	0
resistor	591	18	2	32	0
rsi	328	0	27	22	0
text2wav	488	3	6	38	3
topten	443	67	0	18	0
watertype	284	13	36	14	3
watchimer	947	18	7	15	2
TOTAL	4503	156	148	194	25

Figure 7.1: Annotation overhead on JavaScript code

number that need to be edited or inserted manually (Manual). As the table shows, our 4,500-line codebase requires only 25 manual annotations.

In addition to annotations, a retrofitted type system requires refactorings. Some refactorings expose weaknesses—a static type system cannot account for all legitimate untyped code (the "Problems" column). Other refactorings expose bugs and features that the type system willfully eliminates (the "Fixes" column). Each of these refactorings required approximately one line. Therefore, we changed 6.7% of the lines of code.

The following are a representative sample of programming patterns we willfully eliminate and count as Fixes:

- DOM callbacks that do not use their argument are often written without formal arguments. We signal type errors on arity-mismatches (with the exception of the kinds shown in section 4.1.1).
- We require all identifiers to be statically bound (e.g., 59 of the refactorings in "topten" occur because the program does not use `var` to declare variables).
- Some functions, e.g., `setTimeout`, can take either a string that is eval'd or a function as an argument. To improve program security, we disallow strings as arguments to `setTimeout`.
- It is possible to apply arithmetic operators to arbitrary objects. For example, "watchimer"

subtracts `Dates`, since their `valueOf` method returns a number (UNIX time). We require arithmetic operators (excluding `+`) to receive numeric operands. Therefore, we refactor the program to explicitly call `valueOf`.

We easily account for some of the patterns above. However, by rejecting them, our type-checker makes JavaScript behave more like a "normal" programming language. Furthermore, many of these decisions reflect matters of taste, and can easily be reversed to produce a slightly different type-checker. Indeed, when we type-check ADsafe in chapter 8, we need to disable these checks and admit runtime errors (section 8.8).

The refactorings labeled "Problems" expose some trivial implementation omissions that could be addressed with a little more work, as well as some deeper deficiencies in our type system:

- A pattern seen with callbacks is to declare a variable (e.g., `var x = undefined`) outside and initialize it (e.g., `x = 10`) inside the callback. In such cases, neither typing nor flow-analysis can deduce that $x : \mathsf{Num}$. We typically refactor the declaration (e.g., `var x = -1`).
- The `getElementById` function can return arbitrary HTML elements on the page, so its return type is HTMLElement, the type of all elements. However, "catchit" uses `getElementById` to access a `<canvas>`, which has drawing methods. In such cases, we insert a downcast (checked by a contract). A richer type system for the DOM could eliminate these refactorings [84].
- "countdown" initializes an empty object literal, then incrementally adds fields. We do not support this pattern, since our flow analysis is agnostic to objects. However, this pattern is addressed by recency types [43].
- There are various syntactic patterns that our type checker does not accommodate. For example, we do not account for functions that are lifted arbitrarily, we require arrays literals instead of supporting `new Array()`, etc. With a little more elbow-grease, we could add support for these features in our type system.

As the table shows, most problems occur in just a few programs. These refactorings could be addressed by enriching the type system. Alternatively, since many of the refactorings are mechanical, a tool could refactor them, keeping the type system simple. Type system simplicity matters when types are used as documentation.

Chapter 8

Evaluation II: Security

This chapter[1] presents a type-based technique for verifying language-based Web sandboxes. In particular, we detail our verification of the ADsafe Web sandbox. In the previous chapter, we used types to document programs. In this chapter types also specify security properties. We use types in two very distinct ways:

- We annotate and type-check the 2.5 KLOC ADsafe runtime *library.* In contrast, the previous chapter type-checks a collection of *programs.* Our experience supports folklore that library code is very different from program code. The ADsafe library makes heavier use of type abstraction and reflection. Type-checking revealed several bugs in the ADsafe library.

- We also use types to generate a safe sub-language of JavaScript, following the recipe in section 3.4. Unlike that chapter, the sub-language generated in this chapter is not a toy, but a generalization of ADsafe's existing, ad hoc JavaScript sub-language. The generated sub-language had one discrepancy, which turned out to be a bug in ADsafe's sub-language.

We briefly mentioned language-based Web sandboxing chapter 3. This chapter begins detailing what they do and how they work.

[1]This chapter is based on joint work with Spiridon Aristides Eliopoulos and Joe Gibbs Politz [72].

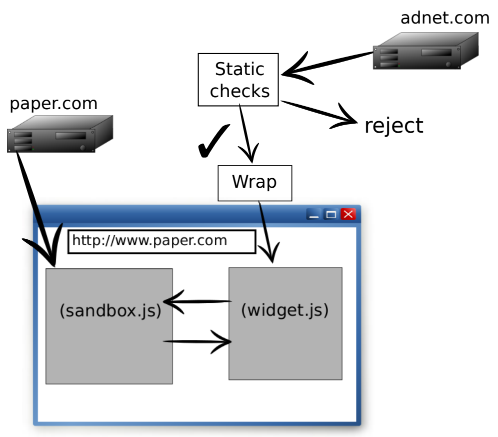


Figure 8.1: Web sandboxing architecture

8.1 Mashups

A *mashup* is a Web page that displays content and executes JavaScript code from various untrusted sources. Facebook applications, gadgets on the iGoogle homepage, and various embedded maps are prominent examples of mashups. Moreover, Web pages that display advertisements from ad networks are also mashups; ads often employ JavaScript for animations and interactivity. A survey of popular pages shows that a large percentage of them include scripts from a diverse array of external sources [94]. Unfortunately, these third-party scripts run with the same privileges as trusted, first-party code served directly from the originating site. Hence, the trusted site is susceptible to attacks by maliciously crafted third-party software.

To address this threat, various organizations have developed *language-based Web sandboxes*. These sandboxes all have similar high-level goals and designs, which we outline in section 8.2. In section 8.3, we review the design and implementation of sandboxes and demonstrate the need for tool-supported verification. Section 8.4 provides a detailed plan for the rest of this chapter.

8.2 Language-based Web Sandboxing

The Web browser environment provides references to objects that implement network access, disk storage, geolocation, and other capabilities. Legitimate web applications use them for various reasons, but embedded widgets can exploit them because all JavaScript on a page runs in the same global environment. A Web sandbox thus attenuates or prevents access to these capabilities, allowing pages to safely embed untrusted widgets. ADsafe [18], Caja [66], FBJS [25], and BrowserShield [73] are *language-based* sandboxes that employ broadly similar security mechanisms, as explained by Maffeis, et al. [59]:

- A Web sandbox includes a static code checker that *filters* out certain widgets that are almost certainly unsafe. This checker is run before the widget is delivered to the browser.
- A Web sandbox provides runtime *wrappers* that attenuate access to the DOM and other capabilities. These wrappers are defined in a trusted runtime library that is linked with the untrusted widget.
- Static checks are necessarily conservative and can reject benign programs. Web sandboxes thus specify how potentially-unsafe programs are *rewritten* to use dynamic safety checks.

This architecture is illustrated in fig. 8.1, where an untrusted widget from `adnet.com` is embedded in a page from `paper.com`. The untrusted widget is filtered by the static checker. If static checking passes, the widget is rewritten to invoke the runtime library. Both the runtime library and the checked, rewritten widget must be hosted on a site trusted by `paper.com`, and are assumed to be free of tampering.

Reference Monitors A Web sandbox implements a *reference monitor* between the untrusted widget and the browser's capabilities. Anderson's seminal work on reference monitors identifies their certification demands [6, p 10-11]:

> The proof of [a reference monitor's] model security requires a verification that the modeled reference validation mechanism is tamper resistant, is always invoked, and cannot be circumvented.

Therefore, a Web sandbox must come with a precisely stated notion of security, and a proof that its static checks and runtime library correctly maintain security. The end result should be a quantified claim of safety over *all* possible widgets that execute against the runtime library.

8.3 Code-Reviewing Web Sandboxes

Imagine we are confronted with a Web sandbox and asked to ascertain its quality. One technique we might employ is a code-review. Therefore, we perform an imaginary review of a Web sandbox, focusing on the details of ADsafe. We then discuss how to (mostly) remove people from the loop.

ADsafe, like all Web sandboxes, consists of two interdependent components:

- a static verifier, called JSLint,[2] which filters out widgets not in a safe subset of JavaScript, and
- a runtime library, `adsafe.js`, which implements DOM wrappers and other runtime checks.

These conspire to make it safe to embed untrusted widgets, though “safe” is not precisely defined. We will return to the definition of safety in section 8.4.

Attenuated Capabilities Widgets should not be able to directly reference various capabilities in the browser environment. Direct DOM references are particularly dangerous because, from an arbitrary DOM reference, `elt`, a widget can simply traverse the object graph and obtain references to all capabilities:

```
var myWindow = elt.ownerDocument.defaultView;
myWindow.XMLHttpRequest;
myWindow.localStorage;
myWindow.geolocation;
```

Widgets therefore manipulate *wrapped* DOM elements instead of direct references. DOM wrappers form the bulk of the runtime library and include many dynamic checks and patterns that need to be verified:

- The runtime manipulates DOM references, but returns them to the widget in wrappers. We must verify that all returned values are in fact wrapped, and that the runtime cannot be tricked into returning a direct DOM reference.
- The runtime calls DOM methods on behalf of the widget. Many methods, such as `appendChild` and `removeChild`, require direct DOM references as arguments. We must verify that the runtime cannot be tricked with a maliciously crafted object that mimics the DOM interface and steals references.
- The runtime attaches DOM callbacks on behalf of the widget. These callbacks are invoked by the browser with event arguments that include direct DOM references. We must verify that the runtime appropriately wraps calls to untrusted callbacks in the widget.

[2]The reader may know that JSLint also performs simple “linting” checks in addition to checks for ADsafe. We only consider JSLint with ADsafe checks enabled.

- The widget has access to a DOM subtree that it is allowed to manipulate. The runtime ensures that the widget only manipulates elements in this subtree. We must verify that various DOM traversal methods, such as `document.getElementById` and `Element.parentNode`, do not allow the widget obtain wrappers to elements outside its subtree.

- The runtime wraps many DOM functions that are only conditionally safe. For example, `document.createElement` is usually safe, unless it is used to create a `<script>` tag, which can load arbitrary code. Similarly, the runtime may allow widgets to set CSS styles, but a CSS URL-value can also load external code. We must verify that the arguments supplied to these DOM functions are safe.

ADsafe's DOM wrappers are called *Bunches*, which wrap collections of HTML elements. There are twenty Bunch-manipulating functions that are exposed to the widget—in addition to several private helper functions—that face all the issues enumerated above and need to be verified. These functions cannot be verified in isolation, because their correctness is dependent on assumptions about the kinds of values they receive from widgets. These assumptions are discharged by the static checks in JSLint and other runtime checks to avoid loopholes and complexities in JavaScript's semantics.

JavaScript Semantics Web sandboxes also contend with JavaScript features that hinder security:

- Certain JavaScript features are unsafe to use in widgets. For example, a widget can use `this` to obtain `window`, so it is rejected by JSLint:

  ```
  f = function() { return this; };
  var myWindow = f();
  ```

  We must verify that the subset of JavaScript admitted by the static checker does not violate the assumptions of the runtime library.

- Many JavaScript operators and functions include implicit type conversions and method calls that are difficult to reason about. For example, when an operator expects a string but is instead given an object, it does not signal an error. Instead, it calls the object's `toString` method. It is easy to write a stateful `toString` method that returns different strings on different calls. Such an object can then circumvent dynamic safety checks that are not carefully written to avoid triggering implicit method calls. These implicit calls are avoided by carefully testing

ADSAFE	:	`ADSAFE.get(obj,name)`
dojox.secure	:	`get(obj,name)`
Caja	:	`$v.r($v.ro('obj'),$v.ro('name'))`
WebSandbox	:	`c(d.obj,d.name)`
FBJS	:	`a12345_obj[$FBJS.idx(name)]`

Figure 8.2: Similar Rewritings for `obj[name]`

the runtime types of untrusted values, using the `typeof` operator. Such tests are pervasive in ADsafe. As a further precaution, ADsafe tries to ensure that widgets cannot define `toString` and `valueOf` fields in objects.

Chapter 2 catalogs these and other JavaScript quirks that can confound the most experienced programmers. Indeed, the bugs we discover (section 8.9) are not deep design errors, but bugs in code to workaround JavaScript's semantic quirks.

JavaScript Encapsulation JavaScript objects have no notion of private fields. If object operations are not restricted, a widget could access built-in prototypes (via the `__proto__` field) and modify the behavior of the container. Web sandboxes statically reject such expressions:

```
obj.__proto__;
```

There are various other dangerous fields that are also *blacklisted* and hence rejected by sandboxes. However, syntactic checks alone cannot determine whether computed field names are unsafe:

```
obj["__pro" + "to__"];
```

Widgets are instead rewritten to use runtime checks that restrict access to these fields. Figure 8.2 shows the rewrites employed by various sandboxes. Some sandboxes insert these and other checks automatically, giving the illusion of programming in ordinary JavaScript. ADsafe requires widget authors to insert the dynamic checks themselves, but the principle remains the same.

Web sandboxes use this method to also simulate private fields. For example, ADsafe stores direct DOM references in the `__nodes__` field of Bunches, and blacklists the `__nodes__` field.

The Reviewability of Web Sandboxes

We have highlighted a plethora of issues that a Web sandbox must address, with examples from ADsafe. Although ADsafe's source follows JavaScript "best practices," the sheer number of checks and abstractions make it difficult to review. There are approximately 50 calls to three kinds of runtime assertions, 40 type-tests, 5 regular-expression based checks, and 60 DOM method calls in the 1,800 LOC `adsafe.js` library. Various ADsafe bugs were found in the past and this chapter presents a few more (section 8.9). Note that ADsafe is a small Web sandbox relative to larger systems like Caja.

The Caja project asked an external review team to perform a code review [7]. The findings describe many low-level details that are similar to those we discussed above. In addition, two higher-level concerns stand out:

- "[Caja is] hard to review. No map states invariants and points to where they are enforced, which hurts maintainability and security."
- "Documentation of TCB is necessary for reviewability and confidence."

These remarks identify an overarching requirement for any review: the need for specifications. Specifications tell us if the system intends to meet our needs. In addition, they help us determine if the implementation is correct.

8.4 Verifying a Sandbox: Our Roadmap

Defining Safety Because humans are expensive and error-prone, and because the code review needs to be repeated every time the program changes, it is best to automate the review process. However, before we begin automating anything, we need some definition of what security means. We focus on a definition that is specific to ADsafe, though the properties are similar to the goals of other web sandboxes. From correspondence with ADsafe's author, we initially obtained the following list of intended properties (rewritten slightly to use the terminology of this chapter).

Definition 2 (ADsafety) *If the containing page does not augment built-in prototypes, and all embedded widgets pass JSLint, then:*

1. *widgets cannot load new code at runtime, or cause ADsafe to load new code on their behalf;*
2. *widgets cannot affect the DOM outside of their designated subtree;*
3. *widgets cannot obtain direct references to DOM nodes; and*
4. *multiple widgets on the same page cannot communicate.*

Note that the first two properties are common to sandboxes in general—allowing arbitrary JavaScript to load at runtime compromises all sandboxes' security goals, and all sandboxes provide mediated access to the DOM by preventing direct access.

We also note that the assumption about built-in prototypes is often violated in practice [27]. Nevertheless, like ADsafe, we make this assumption; mitigating it is outside our scope. Given this definition, our goal is to produce a (mostly) automated verification that supports these properties.

Verifying Safety We employ the general purpose JavaScript type-system developed for this dissertation to define and verify ADsafety. Type-checking is well-suited for this task for several reasons. We choose a static type system as our tool of choice for several reasons. First, programmers are familiar with type systems, and ours is mostly standard (the novelties are detailed in chapters 4 and 5). This lessens the burden on sandbox developers who need to understand what the verification is saying about their code. Second, our type system is much more efficient than most whole-program analyses or model checkers, leading to a quick procedure for checking ADsafe's runtime library (20 seconds). Efficency and understandability allow for incremental use in a tight development loop. Finally, our type system is accompanied by a soundness proof. This property accomplishes the actual verification. Thus, the features of comprehensibility, efficiency, and soundness combine to make type checking an effective tool for verifying some of the properties of web sandboxes.

In order to demonstrate the effectiveness of our type-based verification approach, we use type-based arguments to prove an ADsafety theorem. We mostly achieve this (section 8.8) after fixing bugs exposed by our type checker (section 8.9). The rest of this chapter presents a typed account of untrusted widgets and the ADsafe runtime.

- The ADsafety claim is predicated on widgets passing the JSLint checker. Therefore, we need to model JSLint's restrictions. We do this in section 8.5.
- Once we know what we can expect from JSLint, we can verify the actual reference monitoring code in `adsafe.js` using type-checking (section 8.7).
- Before we can verify `adsafe.js`, we need to account for the details of JavaScript source and model the browser environment in which this code runs. λ_{JS} (chapter 2) tackles most of these details; we discuss its application in section 8.6.

We discuss extensions to verify other Web sandboxes in section 8.10.

8.5 Modeling Secure Sublanguages

All web sandboxes' runtime libraries expect to execute against widgets that have been statically checked and rewritten, as shown in fig. 8.1. These checks and rewrites enforce that widgets are written in a sublanguage of JavaScript. This sublanguage ought to be specified explicitly. We focus here on modeling the checks performed by JSLint, ADsafe's static checker, which presents an interesting challenge: there is no formal specification of the language of JavaScript programs that pass JSLint. Instead, the specification is implicit in the implementation of JSLint itself. In this section, we develop a specification of JSLint-ed widgets.[3]

Only a fraction of JSLint's static checks are related to ADsafe. The rest are `lint`-like code-quality checks. JSLint also checks the static HTML of a widget. Verifying this static HTML is beyond the scope of our work; we do not discuss it further. We instead focus on the security-critical static JavaScript checks in JSLint.

How is JSLint used? The ADsafe runtime makes several assumptions about the shape of values it receives from widgets. These assumptions are not documented precisely, but they correspond to various static checks in JSLint. To model JSLint, we reflect these checks in a *type*, called Widget, which we define below. In section 8.5.2 we discuss how this type relates to the behavior of the JSLint implementation.

8.5.1 A Type for Widgets

We expect that *all variables and sub-expressions* of widgets are typable as Widget. The ADsafe runtime can thus assume that widgets only manipulate Widget-typed values.

Primitives JSLint admits JavaScript's primitive values, with trivial types:

$$\mathsf{Prim} = \mathsf{Num} \cup \mathsf{Str} \cup \mathsf{True} \cup \mathsf{False} \cup \mathsf{Null} \cup \mathsf{Undef}$$

We have separate types for True and False because they are necessary to type-check `adsafe.js` (section 8.7). Prim is an untagged union type, and our type system accounts for common JavaScript

[3]Because we want a strategy that extends to other sandboxes, we do not try to exploit the fact that JSLint is written in JavaScript. The Cajoler of Caja is instead written in Java, and the filters and rewriters for other sandboxes might be written in other languages. The strategy we outline here avoids both getting bogged down in the details of all these languages as well as over-reliance on JavaScript itself.

patterns for discriminating unions. We might initially assume that

$$\mathsf{Widget} \quad = \quad \mathsf{Prim}$$

Objects and Blacklisted Fields JSLint admits object literals but blacklists certain field names as dangerous. All other fields are allowed to contain widget values. We therefore augment the Widget type to include objects. We use *fluid object types* (chapter 5) to describe these objects:

$$\mathsf{Widget} = \quad \mu\alpha.\mathsf{Prim} \cup \mathsf{Ref}\left\{ \begin{array}{l} \overline{\left\{ \begin{array}{l} \texttt{"toString"}, \texttt{"valueOf"}, \\ \texttt{\_\_.*\_\_}, \\ \texttt{"arguments"}, \texttt{"caller"}, \texttt{"callee"}, \cdots \end{array} \right\}}^{\circ} : \alpha, \\ \{\texttt{"toString"}, \texttt{"valueOf"}\} : \mathbf{abs} \end{array} \right\}$$

The type above states that object literals may not have the fields `"toString"` and `"valueOf"`. A tlint-typed object may contain any Widget-typed field, except those in the explicitly blacklisted: all fields that begin and end with two underscores, and a small, fixed collection of fields.

Functions Widgets can create and apply functions, so we must widen our Widget type to admit them. Functions in JavaScript are objects with an internal *code* field, which we add to allowed objects:

$$\ldots \mathsf{Ref}\left\{ \begin{array}{l} \mathit{code} : \mathsf{Global} \cup \alpha \times \alpha \cdots \to \alpha, \\ \ldots \end{array} \right\}$$

The type of the *code* field indicates that widget-functions may have an arbitrary number of Widget-typed arguments and return Widget-typed results.[4] It also specifies that the type of the implicit `this`-argument (written inside brackets) may be either Widget or Global. The type Global is not a subtype of Widget, which expresses the underlying reason for JSLint's rejection of all widgets that contain `this` (see Claim 1 below). If the `this`-annotation is omitted, the type of `this` is $\top$.

Prototypes JSLint does not allow widgets to explicitly manipulate objects' prototypes. However, since field lookup in JavaScript implicitly accesses the prototypes, we specify the type of prototypes

[4]The $\alpha \cdots$ syntax is a literal part of the type, and means the function can be applied to any number of additional α-typed arguments. This is *uniform variable-arity polymorphism* [80].

$$\mathsf{Widget} = \quad \mu\alpha.\mathsf{Prim} \cup \mathsf{Ref}\left\{\begin{array}{l} \overline{\left\{\begin{array}{l} \texttt{"toString"}, \texttt{"valueOf"}, \\ \texttt{\_\_.*\_\_}, \\ \texttt{"arguments"}, \texttt{"caller"}, \texttt{"callee"}, \texttt{"eval"}, \texttt{"prototype"} \\ \texttt{"watch"}, \texttt{"unwatch"}, \texttt{"constructor"} \end{array}\right\}}^{\circ} : \alpha, \\ code : \mathsf{Global} \cup \alpha \times \alpha \cdots \to \alpha, \\ \texttt{"\_\_proto\_\_"} : \begin{array}{l} \mathsf{Object} \cup \mathsf{Function} \cup \mathsf{Bunch} \cup \mathsf{Array}, \\ \cup \mathsf{RegExp} \cup \mathsf{Str} \cup \mathsf{Num} \cup \mathsf{Bool} \end{array} \\ \texttt{"\_\_nodes\_\_"} : \mathsf{Array}\langle\mathsf{HTML}\rangle \cup \mathsf{Undef}, \\ \texttt{"\_\_star\_\_"} : \mathsf{Bool} \cup \mathsf{Undef}, \\ \{\texttt{"toString"}, \texttt{"valueOf"}\} : \mathbf{abs} \end{array}\right\}$$

Figure 8.3: The Widget type

in Widget:

$$\ldots\mathsf{Ref}\left\{\begin{array}{l} \texttt{"\_\_proto\_\_"} : \mathsf{Object} \cup \mathsf{Function} \cup \ldots, \\ \ldots \end{array}\right\}$$

The *proto* field enumerates several safe prototypes, but notably omits DOM prototypes such as `HTMLElement`, since widgets should not obtain direct references to the DOM.

Typing Private Fields Widgets cannot create fields that begin and end with underscores. However, ADsafe uses some of these fields itself as private fields to build the Bunch abstraction:

$$\ldots\mathsf{Ref}\left\{\begin{array}{l} \texttt{"\_\_nodes\_\_"} : \mathsf{Array}\langle\mathsf{HTML}\rangle \cup \mathsf{Undef}, \\ \texttt{"\_\_star\_\_"} : \mathsf{Bool} \cup \mathsf{Undef}, \\ \ldots \end{array}\right\}$$

Notably the type of `"__nodes__"` $\mathsf{Array}\langle\mathsf{HTML}\rangle \cup \mathsf{Undef}$ is not a subtype of Widget, so widgets themselves cannot access it.

The full Widget type in fig. 8.3 is a formal specification of the shape of values that `adsafe.js` receives from and sends to widgets. This type is central to our verification of `adsafe.js` and of JSLint.

8.5.2 Widget and JSLint Correspondence

Though we have offered intuitive arguments for why Widget corresponds to the checks in JSLint, we would like to gain confidence in its correspondence with the behavior of the actual JSLint program that sites use:

Claim 3 (Linted Widgets Are Typable) *If JSLint (with ADsafe checks) accepts a widget e, then e and all of its variables and sub-expressions can be* Widget*-typed.*

We validate this claim by testing. We use ADsafe's sample widgets as positive tests—widgets that should be typable and lintable—and our own suite of negative test cases (widgets that should be untypable and unlintable).[5] Note the direction of the implication: an unlintable widget may still be typable, since our type checker admits safe widgets that JSLint rejects.[6] The type checker could be used as a replacement for JSLint's ADsafe checks, but these tests give us confidence that checking the Widget type corresponds to what JSLint admits in practice.

8.6 Modeling JavaScript and the Browser

Verification of a Web sandbox must account for the idiosyncrasies of JavaScript. It also needs to model the run-time environment—provided by the browser—in which the sandboxed code will execute. Here we discuss how we model the language and the browser.

JavaScript Semantics We use λ_{JS} (chapter 2), which reduces JavaScript to a core semantics, which omits many of JavaScript's complexities. As chapter 2 explains, λ_{JS} is accompanied by a *desugaring* function that maps all JavaScript programs (idiosyncrasies included) to behaviorally equivalent λ_{JS} programs. The transformation explicates much of JavaScript's implicit semantics. Hence, we find it easier to build tools that analyze the much smaller λ_{JS} language than to directly process JavaScript. Section 2.2 argues that the translation to λ_{JS} is adequate by testing it against real world implementations. This testing strategy, and the simplicity of implementation that λ_{JS} enables, give us confidence that our tools correctly account for JavaScript.

Modeling the Browser DOM ADsafety claims that `window.eval` is not applied. To validate this claim, the environment does not ascribe a type to `eval` and related functions such as `document.write`. Finally, certain functions, such as `setTimeout`, behave like `eval` when given strings as arguments. ADsafe does need to call these functions, but it is careful to never call them with strings. In our type environment, we give them restrictive types that disallow string arguments.

[5]Testing revealed a security vulnerability in JSLint (section 8.9.1).
[6]Appendix A.1 has examples of the differences.

```
{
  setTimeout: (Widget → Widget) × Widget → Int,
  document: {
    ...
  },
  ...
}
```

Figure 8.4: A Fragment of the Type of `window`

```
var dom = {
  append:
  function(bunch)
  /*: [Widget ∪ Global]Widget × Widget ··· → Widget */
    { // body of append ... },
  combine:
  function(array)
  /*: [Widget ∪ Global]Widget × Widget ··· → Widget */
    { // body of combine... },
  q:
  function (text)
  /*: [Widget ∪ Global]Widget × Widget ··· → Widget */
    { // body of q... },
  // ... more dom ...
};
```

Figure 8.5: Annotations on the `dom` object

Figure 8.4 specifies a fragment of the type of `window`, which carefully specifies the type of unsafe functions in the environment. The remaining safe DOM does not need to be fully specified. `adsafe.js` only uses a small subset of the DOM methods. These methods require types. The browser environment is therefore modeled with 500 lines of object types (appendix E). This type environment is essentially the specification of foreign DOM functions imported into JavaScript.

8.7 Verifying the Reference Monitor

In section 8.5, we discussed modeling the sublanguage of widgets interacting with the sandboxing runtime. In the case of ADsafe and JSLint, we built up the Widget type as a specification of the kinds of values that the reference monitor, `adsafe.js`, can expect at runtime. In this section, we discuss how we use the Widget type to model the boundary between reference monitor and widget code, and ensure that the runtime library correctly guards critical behavior.

The Widget type specifies the shape of widget values that the ADsafe runtime manipulates. Widget is therefore used pervasively in our verification of `adsafe.js`. For example, consider a typical Bunch method:

```
Bunch.prototype.append = function(child) {
  reject_global(this);
  var elts = child.__nodes__;
  ...
  return this;
}
```

The Bunch objects that ADsafe passes to the widget have `Bunch.prototype` as their *proto* (see fig. 8.3), making these methods accessible. Their use in the widget is constrained only by JSLint, so we must type-check these methods with (only) JSLint's assumptions in mind.

For example, we might assume that the `child` argument above should be a Bunch, the implicit `this` argument should also be a Bunch, and it therefore returns a Bunch. However, JSLint does not provide such strong guarantees. Consider this example, which passes JSLint:

```
var func = someBunch.append;
func(900, true, "junk", -7);
```

Here, `this` is bound to `window`, `child` is a number, and there are additional arguments. Therefore, we cannot assume that `append` has the type $[\mathsf{Bunch}]\mathsf{Bunch} \rightarrow \mathsf{Bunch}$. Instead, the most precise type we can ascribe is:

$$[\mathsf{Widget} \cup \mathsf{Global}]\mathsf{Widget} \cdots \rightarrow \mathsf{Widget}$$

That is, `this` could be Widget-typed or the type of the global object, Global, and the other arguments may have any subtype of Widget, which includes strings, numbers, and other non-Bunch types. The runtime check in `append`'s body (namely, `reject_global(this)`) is responsible for checking that `this` is not the global object before manipulating it. Our type checker recognizes such checks and narrows the broader type to Widget after appropriate runtime checks are applied. If such checks were missing, the type of `this` would remain $\mathsf{Widget} \cup \mathsf{Global}$, and `return this` would signal a type error because $\mathsf{Widget} \cup \mathsf{Global}$ is not a subtype of the stated return type Widget.

Ascribing types to functions provided by the ADsafe runtime is therefore trivial. We give all the same type:

$$[\mathsf{Widget} \cup \mathsf{Global}]\mathsf{Widget} \cdots \rightarrow \mathsf{Widget}$$

The type checker we extend is not ADsafe-specific, and requires explicit type annotations. However, since all the annotations are identical, they are trivial to insert. Figure 8.5 shows a small excerpt of such annotations, which the checker reads from comments, so programs can run unaltered in the browser.

Types for Private Functions ADsafe also has a number of private functions, which are not exposed to the widget. These functions have types with capabilities the widget does not have access to, such as HTML. For example, ADsafe specifies a `hunter` object, which contains functions that traverse the DOM and accumulate arrays of DOM nodes. These functions all have the type $\mathsf{HTML} \rightarrow \mathsf{Undef}$, and add to an array `result` that has type $\mathsf{Array}\langle\mathsf{HTML}\rangle$. ADsafe can freely use these capabilities inside the library as long as it doesn't hand them over to the widget. Our annotations show that it doesn't, because these types are not compatible with Widget.

Type-Checking ADsafe We type-check ADsafe using the JavaScript type-checker developed for this dissertation. ADsafe heavily exercises both flow typing (chapter 4) and fluid object types (chapter 5), the two type-checking innovations that we implement in the type-checker.

The Widget-type, which describes the reference monitor's interface to widgets, evidently uses fluid object types. Widget is an untagged union, and within the reference monitor there are a number of runtime checks to discriminate Widget-typed values. These runtime checks ensure that protected objects—DOM objects and browser functions in ADsafe's case—are only manipulated in safe and well-defined ways. For example, when `setTimeout`'s first argument is a string, rather than a function, it exhibits `eval`-like behavior, which violates ADsafety's constraints. Thus we instead give it the type

$$(\mathsf{Widget} \rightarrow \mathsf{Widget}) \times \mathsf{Widget} \rightarrow \mathsf{Num}$$

Doing so forces the first argument to be a function and, in particular, not a string. Now consider its use:

```
later: function (func, timeout)
/*: Widget × Widget → Widget */ {
  if (typeof func === "function") {
    setTimeout(func, timeout || 0);
  } else { error(); }
}
```

Because `ADSAFE.later` is exported to widgets, it can only assume the Widget type for its arguments, including `func`. A traditional type checker would thus conclude that `func` has type Widget everywhere in `later`. Because Widget includes Str, the invocation of `setTimeout` would yield a type error—even though this is precisely what the conditional in `later` is avoiding!

This is an *if-splitting* pattern that we type-checking using flow typing (chapter 4) flow analysis. The analysis informs the type checker that due to the `typeof` check, uses of `func` in the `then`-branch of the conditional can in fact be *refined* from the large Widget type of $\mathsf{Str} \cup \mathsf{Num} \cup \ldots$ to the function type that `setTimeout` requires.

8.7.1 Required Refactorings

Our type system cannot type check the ADsafe runtime as-is; we need to make some simple refactorings. The need for these refactorings does not reflect a weakness in ADsafe. Rather, they are programming patterns that we cannot verify with our type system. To gain confidence that we didn't change ADsafe's behavior, we run ADsafe's sample widgets against our refactored version of ADsafe, and they behave as expected. We describe these refactorings below:

Additional `reject_name` Checks ADsafe uses `reject_name` to check accesses and updates to object properties in `adsafe.js`. If-splitting uses these checks to narrow string set types and type-check object property references. However, ADsafe does not use `reject_name` in every case. For example, it uses a regular expression to parse DOM queries, and uses the result to look up object properties. Because our type system makes conservative assumptions about regular expressions, it would erroneously indicate that a blacklisted field may be accessed. Thus, we add calls to `reject_name` so the type system can prove that the accesses and assignments are safe.

Inlined `reject_global` Checks Most Bunch methods start by asserting `reject_global(this)`, which ensures that `this` is Widget-typed in the rest of the method. Our type system cannot account for such non-local side-effects, but once we inline `reject_global`, if-splitting is able to refine types appropriately (for instance, in the `Bunch.prototype.append` example early in this section).

`makeableTagName` ADsafe's whitelist of safe DOM elements is defined as a dictionary:

```
var makeableTagName =
  { "div": true, "p": true, "b": true, ... };
```

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

function F() {} // only used below

ADSAFE.create =
  typeof Object.create === 'function'
  ? Object.create
  : function(o) {
      F.prototype =
        typeof o === 'object' && o
               ? o : Object.prototype;
      return new F();
    };
```

Figure 8.6: The Unverified Portion of ADsafe

This dictionary omits an entry for `"script"`. The `document.createElement` DOM method creates new nodes. We ensure that `<script>` tags are not created by typing it as follows:

$$\texttt{document.createElement} : (\texttt{"script"})^{-} \rightarrow \mathsf{HTML}$$

ADsafe uses its tag whitelist before calling `document.createElement`:

```
if (makeableTagName[tagName] === true) {
  document.createElement(tagName);
}
```

Our type checker cannot account for this check. We instead refactor the whitelist (a trick noted elsewhere [61]):

```
var makeableTagName =
  { "div": "div", "p": "p", "b": "b", ... };
```

The type of these strings are $(\texttt{"div"})^{+}$, $(\texttt{"p"})^{+}$,$(\texttt{"b"})^{+}$, etc., so that `makeableTagName[tagName]` has type $(\texttt{"div"}, \texttt{"p"}, \texttt{"b"}, \ldots)^{+}$. Since this finite set of strings excludes `"script"`, it now matches the argument type of `createElement`.

8.7.2 Cheating and Unverifiable Code

A complex body of code like the ADsafe runtime cannot be type-checked from scratch in one sitting. We therefore found it convenient to augment the type system with a `cheat` construct that ascribes a given type to an expression without descending into it. We could thus use `cheat` when we encountered an uninteresting type error and wanted to make progress. Our goal, of course, was to ultimately remove every `cheat` from the program.

We were unable to remove two `cheat`s, leaving eleven unverified source lines in the 1,800 LOC ADsafe runtime. We can, in fact, ascribe interesting types to these functions, but checking them is beyond the power of our type system. Figure 8.6 shows these eleven unverified lines, and we discuss them below.

`reject_name` The `reject_name` function returns `true` if its argument is a blacklisted field and `false` otherwise. This function is used as a predicate to guard against invalid field accesses, so we ascribe it an intersection type:

$$\mathsf{UnsafeField} \rightarrow \mathsf{True} \cap \overline{\mathsf{UnsafeField}} \rightarrow \mathsf{Bool}$$

where UnsafeField is the set of blacklisted field names. Our implemented type-checker does not support checking functions with intersection types, but we type-check applications of intersection-typed functions in the usual way.

`ADSAFE.create` In the ECMAScript 5 standard, `Object.create` takes an object `o` as a parameter and creates a new object whose prototype is `o`; if `o` is not an object, the new object's prototype is `Object.prototype`. ADsafe provides this same functionality for current browsers through `ADSAFE.create`. This function is never used by ADsafe; it is only intended for widgets. Therefore, its type must be

$$[\mathsf{Global} \cup \mathsf{Widget}]\mathsf{Widget} \cdots \rightarrow \mathsf{Widget}$$

JSLint ensures that the actual argument is Widget-typed (section 8.5). However, the return type is problematic. In our Widget type (fig. 8.3), the *proto* field admits Object but not Widget, which is necessary to type-check the code. Permitting α (which represents Widget) in the type of *proto* results in a type system that we have not been able to show will terminate.

`ADSAFE._intercept` ADsafe allows the hosting Web page to define *interceptors*, which are functions that get direct access to the DOM. Verifying interceptors entails verifying that the trusted container is safe. We can do so using the same technology we use to verify the ADsafe runtime—our typechecker!

8.8 ADsafety Redux

Sections 8.5 and 8.7 gave the details of our strategy for modeling JSLint and verifying `adsafe.js`. In this section, we combine these results and relate it to the original definition of ADsafety (definition 2). The use of a type system allows us to make straightforward, type-based arguments of safety for the components of ADsafe.

The lemmas below formally reason about type-checked widgets. Claim 3 (section 8.5.2) establishes that linted widgets are in fact typable. Therefore, *we do not need to type-check widgets.* Widget programmers can continue to use JSLint and do not need to know about our type checker. However, given the benefits of uniformity provided by a type checker over ad hoc methods like JSLint (section 8.9 details one exploit that resulted from such an ad hoc approach), programmers may be well served to use our type checker instead.

Type Soundness We type-check ADsafe using the JavaScript type-checker built for this dissertation. However, we disable all checks for runtime errors. Runtime errors are perfectly acceptable (they halt execution before something bad happens).

Since we admit runtime errors, our type soundness theorem is slightly peculiar. As usual, type soundness is composed of two lemmas: progress and preservation. The preservation lemma is conventional, but the progress lemma does not establish the absence of runtime errors. The progress lemma does however establish that the semantics always signals errors instead of getting stuck. Untyped progress was established in section 2.2.

Our assumed environment (section 8.6) provides the abstract heap Σ and abstract environment Γ, which model the initial state of the browser, σ. We can thus make type-based statements about the combination of widgets and `adsafe.js`:

Theorem 6 (ADsafety) *For all widgets p, if*

1. *all subexpressions of p are Widget-typable,*

2. `adsafe.js` is typable,

3. `adsafe.js` runs before p, and

4. $\sigma p \rightarrow \sigma' p'$ (single-step reduction),

then at every step p', p' also has the type Widget.

Proof: Follows from the type preservation of flow typing (theorem 3) and fluid object types (theorem 4). ■

This theorem says that for all widgets p whose subexpressions are Widget-typed, if `adsafe.js` type-checks and runs in the browser environment, p can take any number of steps and still have the Widget type. Since types are preserved, two further key lemmas hold during execution:

Corollary 1 (Widgets cannot load new code at runtime) *For all widgets e, if all variables and sub-expressions of e are Widget-typed, then e does not load new code.*

By section 8.6, `eval`-like functions are not ascribed types, hence cannot be referenced by widgets or by the ADsafe runtime. Furthermore, functions that only `eval` when given strings, such as `setTimeout`, have restricted types that disallow `string`-typed arguments. Therefore, neither the widget nor the ADsafe runtime can load new code. ■

Corollary 2 (Widgets do not obtain DOM references) *For all widgets e, if all variables and sub-expressions of e are Widget-typed, then e does not obtain direct DOM references.*

The type of DOM objects is not subsumed by the Widget type. All functions in the ADsafe runtime have the type:

$$[\mathsf{Widget} \cup \mathsf{Global}]\mathsf{Widget} \cdots \rightarrow \mathsf{Widget}$$

Thus, functions in the ADsafe runtime do not leak DOM references, as long as they are only applied to Widget-typed values. Since all subexpressions of the widget e are Widget-typed, all values that e passes to the ADsafe runtime are Widget-typed. By the same argument, e cannot directly manipulate DOM references either. ■

Widgets can only manipulate their DOM subtree We cannot prove this claim with our tools. JSLint enforces this property by also verifying the static HTML of widgets; it ensures that all element IDs are prefixed with the widget's ID. The wrapper for `document.getElementById` ensures that the widget ID is a prefix of the element ID. Verifying JSLint's HTML checks is beyond the scope of this work.

In addition, the wrapper for `Element.parentNode` checks to see if the current element is the root of the widget's DOM subtree. It is not clear if our type checker can express this property without further extensions.

Widgets cannot communicate This claim is false; section 8.9.3 presents a counterexample.

8.9 Bugs Found in ADsafe

We have implemented the type system presented in this chapter, and applied it to the ADsafe source. The implementation is about 3,000 LOC, and takes 20 seconds to check `adsafe.js` (mainly due to the presence of recursive types). In some cases, type-checking failed due to the weakness of the type checker; these issues are discussed in section 8.7.1. The other failures, however, represent genuine errors in ADsafe that were present in the production system. The same applies to instances where JSLint and our typed model of it failed to conform. All the errors listed below have been reported, acknowledged by the author, and fixed.

8.9.1 Missing Static Checks

JSLint inadvertently allowed widgets to include underscores in quoted field names. In particular, the following expression was deemed safe:

```
fakeBunch = { "__nodes__": [ fakeNode ] };
```

A malicious widget could then create an object with an `appendChild` method, and trick the ADsafe runtime into invoking it with a direct reference to an HTML element, which is enough to obtain `window` and violate ADsafety:

```
fakeNode = {
  appendChild: function(elt) {
    myWindow = elt.ownerDocument.defaultView;
  }
};
```

```
ADSAFE.go("AD_", function (dom, lib) {
  var myWindow, fakeNode, fakeBunch, realBunch;

  fakeNode = {
    appendChild: function(elt) {
       myWindow = elt.ownerDocument.defaultView;
    },
    tagName: "div",
    value: null
  };

  fakeBunch = {"__nodes__": [fakeNode]};

  realBunch = dom.tag("p");
  fakeBunch.value = realBunch.value;
  fakeBunch.value(""); // calls phony appendChild

  myWindow.alert("hacked");
});
```

Figure 8.7: Exploiting JSLint

The full exploit is in fig. 8.7.

This bug manifested itself as a discrepancy between our model of JSLint as a type checker and the real JSLint. Recall from section 8.5 that all expressions in widgets must have type Widget (defined in fig. 8.3). For `{ "__nodes__": [fakeNode] }` to type as Widget, the `"__nodes__"` field must have type Array⟨HTML⟩∪Undef. However, `[fakeNode]` has type Widget, which signals the error.

JSLint similarly allowed `"__proto__"` and other fields to appear in widgets. We did not investigate whether they can be exploited as above, but setting them causes unanticipated behavior. Fixing JSLint was simple once our type checker found the error. (An alternative solution would be to use our type system as a replacement for JSLint.) We note that when the ADsafe option of JSLint was first announced,[7] its author offered:

> If [a malicious client] produces no errors when linted with the ADsafe option, then I will buy you a plate of shrimp.

We have obtained this shrimp bounty (seven plates of shrimp).

8.9.2 Missing Runtime Checks

Many functions in `adsafe.js` incorrectly assumed that they were applied to primitive strings. For example, `Bunch.prototype.style` began with the following check, to ensure that widgets do not programmatically load external resources via CSS:

[7] `tech.groups.yahoo.com/group/caplet/message/44`

```
ADSAFE.go("AD_", function (dom, lib) {
  var called = false;
  var obj = {
    "toString": function() {
      if (called) {
        return "url(evil.xml#exp)";
      }
      else {
        called = true;
        return "dummy";
      }
    }
  };
  dom.append(dom.tag("div"));
  dom.q("div").style("MozBinding", o);
});

<!-- evil.xml -->
<?xml version="1.0"?>
<bindings><binding id="exp">
<implementation><constructor>
document.write("hacked")
</constructor></implementation>
</binding></bindings>
```

Figure 8.8: Firefox-specific Exploit for ADsafe

```
Bunch.prototype.style = function(name, value) {
  if (/url/i.test(value)) { // regex match?
    error();
  }
  ...
};
```

Thus, the following widget code would signal an error:

```
someBunch.style("background",
  "url(http://evil.com/image.jpg)");
```

The bug is that if `value` is an object instead of a string, the regular-expression `test` method will inadvertently invoke `value.toString()`.

A malicious widget can construct an object with a stateful `toString` method that passes the test when first applied, and subsequently returns a malicious URL. In Firefox, we can use such an object to load an XBL resource[8] that contains arbitrary JavaScript (fig. 8.8).

We ascribe types to JavaScript's built-ins to prevent implicit type conversions. Therefore, we require the argument of `Regexp.test` to have type Str. However, since `Bunch.prototype.style` can be invoked by widgets, its type is $\mathsf{Widget} \times \mathsf{Widget} \to \mathsf{Widget}$, and thus the type of `value` is Widget.

[8]`https://developer.mozilla.org/en/XBL`

This bug was fixed by adding a new `string_check` function to ADsafe, which is now called in 18 functions. All these functions are not otherwise exploitable, but a missing check would cause unexpected behavior. The fixed code is typable.

8.9.3 Counterexamples to Non-Interference

Finally, a type error in `Bunch.prototype.getStyle` helped us generate a counterexample to ADsafe's claim of widget noninterference (definition 2, part 4). The `getStyle` method is available to widgets, so its type must be Widget $\rightarrow$ Widget. The following code is the essence of `getStyle`:

```
Bunch.prototype.getStyle = function (name) {
  var sty;
  reject_global(this);
  sty = window.getComputedStyle(this.__node__);
  return sty[name];
}
```

The bug above is that `name` is unchecked, so it may index arbitrary fields, such as `__proto__`:

```
someBunch.getStyle("__proto__");
```

This gives the widget a reference to the prototype of the browser's `CSSStyleDeclaration` objects. Thus the return type of the body is not Widget, yielding a type error.

A widget cannot exploit this bug in isolation. However, it can replace built-in methods of CSS style objects and interfere with the operation of the hosting page and other widgets that manipulate styles in JavaScript.

This bug was fixed by adding a `reject_name` check that is now used in this and other methods. Despite the fix, ADsafe still cannot enforce non-interference, since widgets can reference and affect properties of other shared built-ins:

```
var arr = [ ];
arr.concat.channel = "shared data";
```

The author of ADsafe pointed out the above example and retracted the claim of non-interference.

Prior Exploits Before and during our implementation, other exploits were found in ADsafe and reported [59–61]. We have run our type checker on the exploitable code, and our tools catch the bugs and report type errors.

Fixing Bugs and Tolerating Changes Each of our bug reports resulted in several changes to the source, which we tracked. In addition to these changes, `adsafe.js` also underwent non-security related refactorings during the course of this work. Even though we did not provide our type checker to its author, we easily continued type-checking the code after these changes. One change involved adding a number of new `Bunch` methods to extend the API. Keeping up-to-date was a simple task, since all the new `Bunch` methods could be quickly annotated with the Widget type and checked. In short, our type checker has shown robustness in the face of program edits.

8.10 Beyond ADsafe

We employed flow typing (chapter 4) to type-check idiomatic JavaScript, but we could have instead employed some other if-splitting technique. However, fluid object types (chapter 5) are instrumental for describing ADsafe's interfaces: ADsafe reasons about infinite collections of fields in uniform ways, which is exactly what fluid object types describe.

Our type-based strategy provides a concrete roadmap for sandbox designers:

1. Formally specify the language of widgets using a type system;
2. use this specification to define the interface between the sandbox and untrusted code; and,
3. check that the body of the sandbox adheres to this interface by type-checking.

In particular, developers of *new* sandboxes should be aware of this strategy. Rather than trying to retrofit the type system's features onto existing static checks, the sandbox designer can work with the type system to guarantee safety constructively from the start. Tweaks and extensions to the type system are certainly possible—for example, one may want to design a sandboxing framework that forbids applying non-function values and looking up fields of `null`, which the current type system allows (section 8.8).

ADsafe shares many programming patterns with other Web sandboxes (section 8.3), but doesn't cover the full range of their features. We outline some of the extensions that could be used to verify them here:

Reasoning About Strings Our type system lets programmers reason about finite sets of strings and use these sets to lookup fields in objects. To verify Caja, we would need to reason about string patterns. For example, Caja uses the field named `"foo"+ "_w__"` to store a flag that determines if the field `"foo"` is writable. We can easily express this and the 8 other patterns used by Caja using fluid object types:

$$\{\texttt{.*\_w\_\_}^{\circ} : \textsf{Bool}, \overline{\texttt{.*\_w\_\_}}^{\circ} : T\}$$

Abstracting Runtime Tests Our type system accounts for inlined runtime checks, but requires some refactorings when these checks are abstracted into predicates. Larger sandboxes, like Caja, have more predicates, so refactoring them all would be infeasible. We could instead use ideas from occurrence typing [86], which accounts for user-defined predicates.

Modeling the Browser Environment ADsafe wraps a small subset of the DOM API and we manually check that this subset is appropriately typed in the initial type environment. This approach does not scale to a sandbox that wraps more of the DOM. If the type environment were instead derived from the C++ DOM implementation, we would have significantly greater confidence in our environmental assumptions.

8.11 Related Work

JavaScript Web Sandboxes ADsafe [18], BrowserShield [73], Caja [66], and FBJS [25] are archetypal Web sandboxes that use static and dynamic checks to safely host untrusted widgets. However, the semantics of JavaScript and the browser environment conspire to make JavaScript sandboxing difficult (chapter 2).

Maffeis et al. [59] use their JavaScript semantics to develop a miniature sandboxing system and prove it correct. Armed with the insight gained by their semantics and proofs, they find bugs in FBJS and ADsafe (which we also catch). However, they do not mechanically verify the JavaScript code in these sandboxes. They also formalize capability safety and prove that a Caja-like subset is capability safe [61]. However, they do not verify the Caja runtime or the actual Caja subset. In contrast, we verify the source code of the ADsafe runtime and account for ADsafe's static checks.

Taly, et al. [82] develop a flow analysis to find bugs in the ADsafe runtime (that we also catch). They simplify the analysis by modeling ECMAScript 5 strict mode, which is not fully implemented in any current Web browser. In contrast, ADsafe is designed to run on current browsers, and thus supports older and more permissive versions of JavaScript. We use λ_{JS} and associated tools (chapter 2), which which is not limited to strict mode, so we find new bugs in the ADsafe runtime. In addition, Taly, et al. use a simplified model of JSLint. In contrast, we provide a detailed, type-theoretic account of JSLint, and also test it. We can thus find security bugs in JSLint as well.

Lightweight Self-Protecting JavaScript [62, 70] is a unique sandbox that does not transform or validate widgets. It instead solely uses reference monitors to wrap capabilities. These are modeled as security automata, but the model ignores the semantics of JavaScript. In contrast, this chapter and the aforementioned works are founded on detailed JavaScript semantics.

Yu, et al. [93] use JavaScript sandboxing techniques to enforce various security policies on untrusted code. Their semantic model, CoreScript, simplifies the DOM and scripting language. CoreScript cannot be used to mechanically verify the JavaScript implementation of a Web sandbox, which is what we present in this chapter.

Modeling the Web Browser There are formal models of Web browsers that are tailored to model whole-browser security properties [2, 10]. These do not model JavaScript's semantics in any detail and are therefore orthogonal to semantic models of JavaScript (chapter 2 and [58]) that are used to reason about language-based Web sandboxes. In particular, ADsafe's stated security goals are limited to statements about JavaScript and the DOM (section 8.4). Therefore, we do not require a comprehensive Web-browser model.

Static Analysis of JavaScript GateKeeper [37] uses a combination of program analysis and runtime checks to apply and verify security policies on JavaScript widgets. GateKeeper's program analysis is designed to model more complex properties of untrusted code than we address by modeling JSLint. However, the soundness of its static analysis is proven relative to only a restricted sub-language of JavaScript, whereas λ_{JS} handles the full language. In addition, they do not demonstrate the validity of their run-time checks.

Chugh et al. [15] and VEX [8] use program analysis to detect possibly malicious information flows in JavaScript. Our type system cannot specify information flows, although we do use it to discover that ADsafe fails to enforce a desirable information flow property. VEX's authors acknowledge that it is unsound, and Chugh et al. do not provide a proof of soundness for their flow analysis. Our type system and analysis are proved sound.

Other static analyses for JavaScript [38, 51, 52] are not specifically designed to encode and check security.

Language-Based Security Schneider et al. [77] survey the design and type-based verification of language-based security systems. JavaScript Web sandboxes are inlined reference monitors [24].

Cappos, et al. [14] present a layered approach to building language sandboxes that prevents bugs in higher layers from breaking the abstractions and assurances provided by lower layers. They use this approach to build a new sandbox for Python, whereas we verify an existing, third-party JavaScript sandbox. However, our verification techniques could easily be used from the onset to build a new sandbox that is secure by construction.

IFrames IFrames are widely used for widget isolation. However, JavaScript that runs in an IFrame can still open windows, communicate with servers, and perform other operations that a Web sandbox disallows. Furthermore, inter-frame communication is difficult when desired; there are proposals to enhance IFrames to make communication easier and more secure [50]. Language-based sandboxing is somewhat orthogonal in scope, is more flexible, and does not require changes to browsers.

Runtime Security Analysis of JavaScript There are various means to secure widgets that do not employ language-based security. Some systems rely on modified browsers, additional client software, or proxy servers [20, 21, 53, 54, 57, 64, 93]. Some of these propose alternative Web programming APIs that are designed to be secure. Language-based sandboxing has the advantage of working with today's browsers and deployment methods, but our verification ideas could potentially apply to the design of some of these systems, too.

Chapter 9

Related Scripting Languages

This dissertation focuses on JavaScript; all the software developed to support this dissertation targets JavaScript. This chapter[1] demonstrates that despite syntactic distinctions, the semantics of Python, Ruby, and JavaScript have many common elements. We therefore believe that the type-checking techniques presented in this dissertation (chapters 4 and 5) are more broadly applicable. It is a mere matter of programming to build a core calculus and tested desugaring functions for Python and Ruby, in the style of λ_{JS} (chapter 2).

9.1 Objects, Dictionaries, and Inheritance

Untyped scripting languages implement objects as simple dictionaries, mapping member names to values. Inheritance affects member lookup, but does not affect updates and deletion. This semantics is clear in the following JavaScript program:

```
var parent = { "z": 9 };
var obj = { "x": 1, "__proto__": parent };

obj.x // returns 1
obj.z // returns 9
obj.z = 50 // creates new field in obj
obj.z // returns 50, shadowing parent.z
parent.z // returns 9; parent.z not set by obj.z = 50
```

[1]This chapter is based on joint work with Claudiu Saftoiu [41] and Joe Gibbs Politz [39].

```
class Parent; def z; return 9; end; end

obj = Parent.new
class << obj; def x; return 1; end; end

obj.x # returns 1
obj.z # returns 9
class << obj; def z; return 50; end; end
obj.z # return 50
# no simple way to invoke shadowed z method
class << obj; remove_method :z; end
obj.z # returns 9

class << obj
  define_method("xyz".to_sym) do; return 99; end
end

print obj.xyz # returns 99
```

Figure 9.1: Changing Object Shapes in Ruby

This program creates two objects, `obj` and `parent`, where `obj` inherits from `parent`. In Python, we cannot set up inheritance directly; however, the following program creates an equivalent object graph:

```
class parent(object)
  z = 9 # class member
  def __init__(self): self.x = 1 # instance member

obj = parent()

obj.x # returns 1
obj.z # returns 9
obj.z = 50 # creates new field
obj.z # returns 50, shadowing parent.z
parent.z # returns 9, just like JavaScript
```

We can delete the `parent.z` field, returning both programs to their initial state, in JavaScript—

```
delete obj.z
obj.z // returns 9
```

—and in Python—

```
delattr(obj, "z")
obj.z # returns 9
```

Finally, field names are simply strings and do not need to be statically specified in JavaScript—

```
obj["x " + "yz"] = 99 // creates new field
obj["x y" + "z"] // returns 99
```

—and in Python—

```
setattr(obj, "x " + "yz", 99) # creates new field
getattr(obj, "x y" + "z") # returns 99
```

Figure 9.1 shows that we can translate this sequence to Ruby with only a little more syntactic effort.

Classes Do Not Shape Objects One consequence of exposing the mutable dictionary underlying objects is that a class determines the members of its instances only during initialization. Members can be subsequently added and deleted from individual objects. There is substantial evidence that this occurs in real JavaScript programs [74]. The examples in section 9.5 illustrate that such behavior also occurs in Python and Ruby.

9.2 Inheritance in Scripting Languages

A more subtle consequence is that inheritance is brittle in scripting languages. Consider the following Ruby class:

```
class A
  def initialize; @privateFld = 90; end

  def myMethod; return @privateFld * @privateFld; end
end
```

Ostensibly, the interface of class `A` is only `myMethod`, and `privateFld` is part of the implementation. Consider the following subclass:

```
class B < A
  def initialize; super(); @privateFld = "my string"; end
end
```

Both `A` and `B` use the name `privateFld` in their implementations. However, since `privateFld` is simply a dictionary key, `B` obliterates assumptions about the implementation of `A`:

```
obj = B.new
B.myMethod # error: cannot multiply strings
```

```
class A(object):
  def method(self): return "from class A"

class B(object):
  def method(self): return "from class B"

obj = A()
obj.method() # returns "from class A"
isinstance(obj, A) # returns True

obj.__class__ = B
obj.method() # returns "from class B"
isinstance(obj, B) # returns True: class changed!
```

Figure 9.2: Fluid Class Hierarchies in Python

It is not safe to subclass `A` without knowing its internals. Indeed, the principal author of Ruby declares that "it is only safe to extend Ruby classes when you are familiar with (*and in control of*) the implementation of the superclass" [32, page 240] (emphasis added).

In Python, member names that begin with two underscores are prefixed with the name of the enclosing class. For example, a member named `__field__` is renamed to `__MyClass_field__` within `MyClass`. This syntactic trick is supposed to preserve the modularity of classes. However, when used with modules, if `A.MyClass` extends `B.MyClass`, the module name is not used in the prefix. Thus the problem demonstrated in Ruby also occurs in Python.

9.3 Classes and Prototypes

The previous examples showed how objects' shapes can be altered in scripting languages. Since classes are objects themselves, they can be altered as well. Python and JavaScript go further and expose the inheritance hierarchy as mutable members. Assigning to `obj.__super__` in Python and `obj.__proto__` in JavaScript affects the inheritance chain (fig. 9.2).

9.4 Methods?

JavaScript simply does not have methods. The `obj.method(...)` syntax binds `obj` to an implicit argument, `this`, that is supplied to the function `obj.method` [23, Section 11.2.3]. However, `obj.method` is not associated with either `obj` or its prototype. We can extract the underlying function and call it:

```
var f = obj.method; f()
```

Since `f()` does not use method call syntax, it is treated as a function call; `this` is not bound to `obj`, but to a default "global object".

Unlike JavaScript, Python and Ruby make it harder to extract methods. However, their methods are still unlike methods in other object-oriented languages. For example, consider Python methods, which have an explicit `self` argument:

```
class A(object):
  def __init__(self): self.myField = 900

  def method(self): return self.myField
```

These methods can be extracted from objects and treated as functions that are partially applied to the appropriate `self` argument:

```
obj = A() # construct
f1 = obj.method # extract
f1() # apply, returns 900
```

Re-extracting a method returns the same reference:

```
f2 = obj.method # the same function?
f1 == f2 # returns True--same function
```

Extracting `method` from a different instance results in a distinct reference, which is closed over that instance as the `self` argument:

```
obj2 = A()
obj.method == obj2.method # returns False
```

However, we can still obtain the underlying function, called an *unbound method* in Python, which expects an explicit instance for the `self` argument:

```
f_orig = obj.method.im_func # im_func is built-in
f_orig() # Error: expected 1 argument
f_orig(obj2) # returns 900
obj.method.im_func == obj2.method.im_func # True
```

Ruby methods can similarly be extracted and applied.

Checks For	JS Gadgets	Python stdlib	Ruby stdlib	Django	Rails
`undefined/null` [a]	3,298	1,686	538	868	712
`instanceof` [b]	17	613	1,730	647	764
`typeof` [c]	474	381		4	
field-presence [d]		504	171	348	719
Total Checks	3,789	3,184	2,439	1,867	2,195
LOC	617,766	313,938	190,002	91,999	294,807

[a] `None` in Python, and `nil` in Ruby
[b] `isinstance` in Python, and `.is_a?` and `.instance_of?` in Ruby
[c] `type` in Python
[d] `hasattr` in Python, and `.respond_to?` in Ruby

Figure 9.3: Tag Checks and Related Checks

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

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

Figure 9.4: Reflection APIs

9.5 Reflection and Pattern Matching

Reflection is not unique to scripting languages; the JVM and the .NET CLI have powerful reflection APIs. However, to use their reflection APIs, programmers must use many explicit downcasts.[2] Reflection is significantly easier to use in untyped scripting languages and thus more prevalent.

Figure 9.4 classifies several common object-related reflective operators found in scripting languages. Programs can reflect on both an object's class (nominal) and its members (structural), since an object's class does not fully determine its members. In contrast, Java only supports nominal reflection. `Object.getClass` does allow Java programs to reflect on the structure of classes. However, because classes fully determine objects' members, Java does not have operations to examine the structure of individual instances.

Figure 9.3 offers a conservative estimate of the prevalence of type tests and related checks across a broad corpus JavaScript, Python, and Ruby code, by counting occurrences of type testing operators. We believe these numbers undercount, since they do not account for heap-sensitive reasoning and other type testing patterns. For example, we do not try to estimate how often JavaScript programs

[2] C# 4 adds a `dynamic` type that elaborates to use the reflection API [9].

```
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 9.5: Banned Check from ADsafe

test for the presence of a field, because this operation is syntactically indistinguishable from field lookup.

Sandboxes Reflection in scripting languages is particularly powerful when combined with pattern matching on member names. For example, JavaScript sandboxes like ADsafe and Caja 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 field-lookup expressions (`obj[name]`) in untrusted code are rewritten to check whether `name` is banned. Figure 9.5 is ADsafe's check; it uses a collection of ad hoc tests and also ensures that `name` is not the name of any field in the `banned` object, which is effectively used as a set of names.

Django The Python Django ORM dynamically builds classes based on database information. In the following snippet, it adds a field `attr_name`, that represents a database column, to a class `new_class`, which it is constructing on-the-fly:[3]

```
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)
```

The computed string, `attr_name`, is not arbitrary; it concatenates `"_ptr"` onto `base._meta.module_name`.

Ruby on Rails When setting up a user-defined model, ActiveRecord iterates over the fields of an object and only processes members that match certain patterns:[4]

```
  attributes.each do |k, v|
```

[3]https://github.com/django/django/blob/master/django/db/models/base.py#L157
[4]https://github.com/rails/rails/blob/master/activerecord/lib/active_record/base.rb#L1717

	JavaScript	Python	Ruby
Loops	✓	✓	✓
Exceptions	✓	✓	✓
Generators		✓	✓
Labelled Statements	✓		
Switch fall-through	✓		
Continuations			✓

Figure 9.6: Control Features of Scripting Languages

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

The first pattern, `k.include?("(")`, checks the shape of the field name `k`, and the second pattern checks if the object has a member called `"#"+ k + "="`.

Java Beans Even in Java, programmers employ reflective patterns. Java Beans provide a flexible component-based mechanism for composing applications. The Java Beans API uses reflective reasoning on canonical naming patterns to construct classes on-the-fly. For example, `java.beans.Introspector` "applies the naming conventions to determine what properties the bean has, the events to which it can listen, and those which it can send."[5] Properties of Beans are not necessarily known at runtime, 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, and require runtime reasoning about casts to use.

9.6 Control Operators

JavaScript, Python, and Ruby have a diverse set of control operators (fig. 9.6). These are all easily modeled in a Felleisen-Hieb style semantics [26]. For the purpose of analysis, it is easier to pick a

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

uniform representation for all operators, such as continuation-passing style, as we do to type-check JavaScript (section 4.5).

9.7 A Scripting Language Object Calculus

The preceding section presents examples that illustrate the characteristic features of objects in scripting languages. We distill these features into λ_{Sc}, a core calculus of functional objects, imperative state, higher-order functions, and reflection. The combination of mutable references and functional objects is sufficient to encode the imperative objects of scripting languages. λ_{Sc} is not, however, intended to be a scripting language itself. It is designed to be a *desugaring* target for a full scripting language. Its syntax and semantics are in fig. 9.7 and introduced incrementally below.

Objects as Dictionaries λ_{Sc} faithfully models the objects-as-dictionaries design of scripting languages.

- In a field lookup, e_1[e_2], the field name is not a static string, but an arbitrary expression e_2, and it is a runtime error if e_2 does not evaluate to a string.

- Expressions of the form e_1[e_2 = e_3], are used to both update the value of existing fields (E-Update) and, if the field does not exist, to create new fields (E-Create). This is the behavior of all scripting languages; fields do not need to be declared and different instances of the same class or prototype can have different sets of fields.

- Fields can be deleted with **delete** e_1[e_2] (E-Delete). If the specified field does not exist, λ_{Sc} silently continues evaluation. This models JavaScript's semantics, whereas other languages signal exceptions; it is routine to adapt the semantics to signal exceptions instead.

- When a field is not found, λ_{Sc} looks for the field in the parent object, which is the value of the parent field (E-Inherit). In JavaScript implementations this field is called "__proto__", and in Python it is called "__super__". In Ruby, the name is not visible to programmers (section 9.3).

Given the lack of syntactic restrictions on object lookup, we can easily write a program that looks up a field that is not defined anywhere on the inheritance chain. In such cases, λ_{Sc} signals an error (E-NotFound). This is a model of Python and Ruby, but not JavaScript. Unusually, JavaScript

$$\begin{array}{rcll}
l &=& \cdots & \text{Locations}\\
P &=& \cdots & \text{String patterns}\\
\sigma &=& \cdot \mid (l,v)\sigma & \text{Stores}\\
c &=& num \mid str \mid bool \mid \mathbf{null} & \text{Constants}\\
v &=& c \mid l \mid \mathbf{func}(x)\ \{\ e\ \} \mid \{\ str{:}v\cdots\ \} & \text{Values}\\
e &=& x \mid v \mid e(e) \mid \{\ str{:}\ e\cdots\ \} \mid e[e] \mid e[e\ =\ e] \mid \mathbf{delete}\ e[e] & \text{Expressions}\\
& \mid & e\ {:=}\ e \mid \mathbf{ref}\ e \mid \mathbf{deref}\ e \mid \mathbf{if}\ (e_1)\ e_2\ \mathbf{else}\ e_3 \mid \mathbf{fieldin}\ e\ \text{init}\ v_{acc}\ \mathbf{do}\ v_f & \\
& \mid & e_1\ \mathbf{hasfield}\ e_2 \mid e\ \mathbf{matches}\ P \mid \mathbf{err} & \\
E &=& \bullet \mid E(e) \mid v(E) \mid \{\ str{:}\ v\cdots\ str{:}E,\ str{:}e\cdots\ \} \mid E[e] \mid v[E] & \text{Evaluation Contexts}\\
& \mid & E[e\ =\ e] \mid v[E\ =\ e] \mid v[v\ =\ E] \mid \mathbf{delete}\ E[e] \mid \mathbf{delete}\ v[E] \mid E\ {:=}\ e & \\
& \mid & v\ {:=}\ E \mid \mathbf{ref}\ E \mid \mathbf{deref}\ E \mid \mathbf{fieldin}\ E\ \text{init}\ v_{acc}\ \mathbf{do}\ v_f \mid \mathbf{if}\ (E)\ e_2\ \mathbf{else}\ e_3 & \\
& \mid & E\ \mathbf{hasfield}\ e \mid v\ \mathbf{hasfield}\ E \mid E\ \mathbf{matches}\ P &
\end{array}$$

$\boxed{e \hookrightarrow e}$

β_v	$(\mathbf{func}(x)\ \{\ e\ \})(v) \hookrightarrow e[x/v]$
E-GetField	$\{\ \cdots str{:}\ v\cdots\ \}[str] \hookrightarrow$ v
E-Inherit	$\{\ str\ :\ v\cdots\ \text{"parent"}{:}\ l\ \}[str_x] \hookrightarrow \mathbf{deref}\ l$, if $str_x \notin (str\cdots)$
E-NotFound	$\{\ str\ :\ v\cdots\ \text{"parent"}{:}\ \mathbf{null}\ \}[str_x] \hookrightarrow \mathbf{err}$, if $str_x \notin (str\cdots)$
E-Update	$\{\ str_1{:}\ v_1\cdots\ str_i{:}\ v_i\ \cdots str_n{:}\ v_n\ \}\ [str_i\ =\ v]$
	$\hookrightarrow \{\ str_1{:}\ v_1\cdots\ str_i{:}\ v\ \cdots str_n{:}\ v_n\ \}$
E-Create	$\{\ str_1{:}\ v_1\cdots\ \}\ [str_x\ =\ v_x] \hookrightarrow \{\ str_x{:}\ v_x,\ str_1{:}\ v_1\cdots\ \}$, when $str_x \notin (str_1\cdots)$
E-Delete	$\mathbf{delete}\ \{\ str_1{:}\ v_1\cdots\ str_x{:}\ v_x\ \cdots str_n{:}\ v_n\ \}\ [str_x] \hookrightarrow \{\ str_1{:}\ v_1\cdots str_n{:}\ v_n\ \}$
E-Delete-Err	$\mathbf{delete}\ \{\ str_1{:}\ v_1\cdots\ \}\ [str_x] \hookrightarrow \{\ str_1{:}\ v_1\cdots\ \}$, if $str_x \notin (str_1\cdots)$
E-FieldIn	$\mathbf{fieldin}\ \{\ str_1{:}v_1,\ str_2 : v_2\cdots\ \}\ \text{init}\ v_{acc}\ \mathbf{do}\ v_f$
	$\hookrightarrow \mathbf{fieldin}\ \{\ str_2 : v_2\ \cdots\ \}\ \text{init}\ v_f(str_1)(v_{acc})\ \mathbf{do}\ v_f$
E-FieldIn-End	$\mathbf{fieldin}\ \{\ str{:}v\ \}\ \text{init}\ v_{acc}\ \mathbf{do}\ v_f \hookrightarrow v_f(str)(v_{acc})$
E-IfTrue	$\mathbf{if}\ (\mathbf{true})\ e_2\ \mathbf{else}\ e_3 \hookrightarrow e_2$
E-IfFalse	$\mathbf{if}\ (\mathbf{false})\ e_2\ \mathbf{else}\ e_3 \hookrightarrow e_3$
E-HasField	$\{\cdots str{:}v\cdots\ \}\ \mathbf{hasfield}\ str \hookrightarrow \mathbf{true}$
E-HasNotField	$\{\ str{:}v\cdots\ \}\ \mathbf{hasfield}\ str \hookrightarrow \mathbf{false}$, when $str' \notin (str\cdots)$
E-Matches	$str\ \mathbf{matches}\ P \hookrightarrow \mathbf{true}$, $str \in P$
E-NoMatch	$str\ \mathbf{matches}\ P \hookrightarrow \mathbf{false}$, $str \notin P$

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

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

$\boxed{wf \vdash e}$

$$\frac{wf \vdash e_1 \cdots wf \vdash e_n \qquad wf \vdash e_p \qquad str_i \text{ all unique} \qquad str_i \neq \text{"parent"}}{wf \vdash \{\ str_1{:}e_1\cdots str_n{:}e_n,\ \text{"parent"}{:}e_p\ \}}$$

The *wf* relation is defined recursively over other expressions and lifted to stores in the natural way.

Figure 9.7: Syntax and Semantics of λ_{Sc}

does not signal an error, but returns the default value `undefined`. To model JavaScript, we can introduce an **undefined** value and employ the following reduction instead of E-NotFound:

$$\texttt{\{}\ str\ \texttt{:}\ v\cdots\ \text{"parent"}\texttt{:}\ \textbf{null}\ \texttt{\}[}str_x\texttt{]} \hookrightarrow \textbf{undefined}$$
$$\text{if } str_x \notin (str\cdots)$$

The complexity of desugaring a particular scripting language is affected by the chosen reduction relation; desugaring JavaScript is simpler with this alternate reduction. Our formal proofs use E-NotFound, but it is easy to use the JavaScript-like reduction instead.

The fragment of λ_{Sc} presented thus far is sufficient to desugar the examples that do not use reflection.

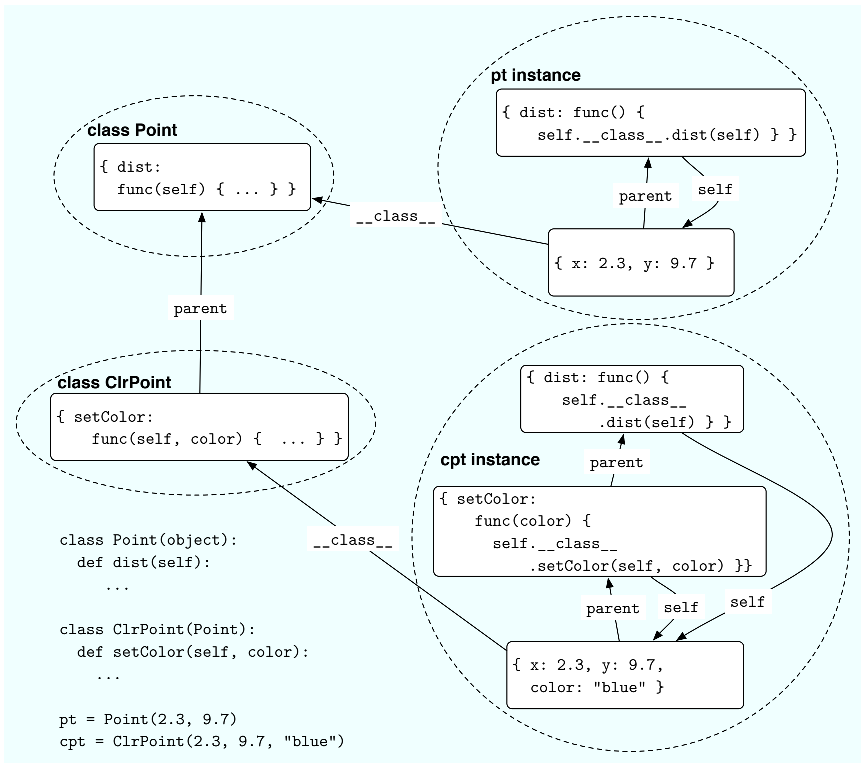
Reflection Nominal reflection in λ_{Sc} is trivial. An object, obj, inherits from a parent, p, if obj["parent"] == p. Structural reflection requires additional operators:

- The *o* **hasfield** *str* expression checks whether an object *o* has a member *str*. This expression is similar to `hasOwnProperty` in JavaScript, **hasfield** does not traverse the inheritance chain to determine if *o* inherits *str*. The behavior of Python's `hasattr` and Ruby's `respond_to?` is easily recovered by desugaring to a loop.

- The **fieldin** *obj* init *init* **do** *f* expression folds the function *f* over the names of the fields of *obj*, with *init* as an accumulator. Unlike the `for in` loop of JavaScript or the `dir` function of Python, **fieldin** does not produce names in the inheritance chain. Actual `for in` loops can be desugared to use **fieldin**.

String Pattern Matching Section 9.5 shows programs that use a variety of operators to pattern match strings that are then used as field names. To model these programs, we introduce an abstract string-matching operator, *str* **matches** *P*. In an actual scripting language, these patterns might be regular expressions. The precise encoding of patterns is irrelevant to our core calculus. We thus abstract P to be an arbitrary class of string-sets with decidable membership.

Classes and Prototypes The classes and prototypes of scripting languages can be desugared to the records of λ_{Sc}. Desugaring JavaScript's prototype inheritance is natural; section 2.1.2 describe


pt instance
{ dist: func() { self.__class__.dist(self) } }
class Point
{ dist: func(self) { ... } }
parent
self
__class__
{ x: 2.3, y: 9.7 }
parent
class ClrPoint
{ setColor: func(self, color) { ... } }
{ dist: func() { self.__class__ .dist(self) } }
cpt instance
parent
{ setColor: func(color) { self.__class__ .setColor(self, color) }}
__class__
parent
self
self
{ x: 2.3, y: 9.7, color: "blue" }

```
class Point(object):
  def dist(self):
    ...

class ClrPoint(Point):
  def setColor(self, color):
    ...

pt = Point(2.3, 9.7)
cpt = ClrPoint(2.3, 9.7, "blue")
```

Figure 9.8: Encoding Python's Method Binding in λ_{Sc}

how `new`, `this`, and other keywords are easily desugared to their semantics; the same strategy applies to λ_{Sc}.

To encode Ruby and Python's classes, one could add primitive classes to λ_{Sc} and model them directly. These primitive classes would need to admit the introspective operations and observations employed by our examples, such as extracting methods, affecting the inheritance hierarchy, and modifying classes. Because the objects of λ_{Sc} naturally admit these observations and effects, we instead study classes by desugaring them. For example, the Python code in fig. 9.8 can be desugared to a λ_{Sc} program that produces the object graph in the same figure.

Each Python object instance is desugared into a collection of λ_{Sc} objects—an object containing the instance's fields (self) and auxiliary objects containing the "bound methods" for each inherited class. This desugaring admits the observations and operations on Python objects made in section 9.4:

- Since bound methods are closed over self, programs can extract them and correctly use them as functions.

- A program can distinguish inherited fields from instance fields.

- A program can redefine a method on a class and have references to existing bound methods call the new method.

The desugaring function that produces the object graph in fig. 9.8 is straightforward: constructor invocation creates the self object, then walks the inheritance chain to create the auxiliary objects containing bound methods, closed over self. Finally, the programmer-written constructor (`__init__` in Python) is applied to self.

9.8 Limitations

λ_{Sc} models the essential features of objects that are common to Ruby, Python, and JavaScript, but omits the following features:

- Python's multiple-inheritance,
- Getters and setters, and
- Object proxies and proxy-like traps in Python and Ruby [19].

These features could be desugared into λ_{Sc}, but it would be clearer to support them directly. We also elide other features that are not directly related to objects, such as eval, concurrency, control operators, and implicit type conversions, as they are not directly related to our goal of studying objects and reflection.

9.9 Soundness

We mechanize λ_{Sc} with PLT Redex [26]. We use our mechanized semantics to test [56] λ_{Sc} for safety.

Theorem 7 (Progress) *If σe is a closed, well-formed configuration, then either:*

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

This property requires additional evaluation rules for runtime errors, which we elide for clarity.

Chapter 10

Conclusions and Future Work

This dissertation demonstrates that *semantics engineering* [26] works in practice. We develop λ_{JS}, which is a Felleisen-Hieb style core calculus for the essentials of JavaScript—a language without any formal specification. We engineer a translation from the full language to the core calculus. Finally, we test that the semantics and translation faithfully model implementations. This process could be repeated for any other programming language.

We use λ_{JS} as the basis of our proofs and our implementations. As of this writing, λ_{JS} is used by research groups at several other institutions, including Microsoft Research, Fujistu Laboratories, Northeastern University, University of Utah, KAIST, and University of Chile.

This dissertation presents two novel type-checking principles:

- *Flow typing* combines type-checking and flow analysis to type-check programs that use local control and state to affect types. We present a principled technique to combine the type-checker and flow analysis in a simple and modular way.

- *Fluid object types* are a natural generalization of record types and row type variables. They give a static account of object systems that support reflection and dynamic field names.

We implement flow typing and fluid object types in a JavaScript type-checker. Using this type-checker, we demonstrate that:

- Types are an effective way to document JavaScript code,

- Types can verify security properties of JavaScript programs, and

- Types can be employed to mechanically generate safe sub-languages of JavaScript.

We also give evidence that our techniques are applicable to other scripting languages.

Future Work

There are various immediate avenues of research that could build on this dissertation.

Semantics λ_{JS} is approximately a semantics for ECMA-262, 3rd edition, which is presently the most widely-used version of JavaScript. JavaScript is evolving rapidly and λ_{JS} needs to keep up with new standards and experimental features. The S5 project[1] at Brown University grows λ_{JS} to support ECMA-262, 5th edition. S5 is better tested than λ_{JS}, using the new standardized test suite for JavaScript, test262.[2] S5 tackles addition features, such as `eval`, that λ_{JS} omits.

The testing strategy we use to demonstrate λ_{JS}'s adequacy can also be used to develop semantics for other scripting languages. As a rule, scripting languages do not bother with formal semantics. But, the languages that matter have canonical implementations and extensive test suites. These two components—an oracle and a set of valid inputs—are all that a good semanticist needs to reverse-engineer a formal semantics. A more interesting problem might be to automate the semantics reverse-engineering process.

Types We use our tools to type-check approximately $7,000$ lines of code. What would it take to increase this number by a few orders of magnitude? Can we it without compromising type soundness? Do we have to weaken the guarantees our type-checker provides? (e.g., must we admit Java-like `NullPointerExceptions`?) We cannot manually annotate and type-refactor millions of lines of code. We need some form of tool support, but it does not have to be ML-style type inference. In section 6.1, we discuss a simple tool that uses instrumentation to infer types that are might be incorrect; such techniques could be pushed further. Future work should be guided by studying type-checking failures with an existing type-checker, such as ours.

Security We verify ADsafe, which is the simplest of the major Web sandboxes. Verifying a more sophisticated sandbox such as Google Caja, which is used to isolate untrusted code by Yahoo! and

[1]`http://www.cs.brown.edu/research/plt/dl/S5`
[2]`http://test262.ecmascript.org`

Google, seems to be the obvious next step. Such a verification would be useful, but it is not clear if it would produce new research results. Types could instead be used to develop a different kind of Web sandbox that addresses some limitations of current approaches:

- The runtime checks in Web sandboxes have an overhead that impedes adoption. Google Caja presently reports 2x or higher slowdowns on various benchmarks.[3] A type-checker can determine that untrusted code does not violate the sandbox's safety properties, and thus safely bypass runtime checks. Crucially, statically unverifiable code would continue to use runtime checks.

  Indeed, in chapter 8, the safe sub-language we generate from types is already less restrictive than ADsafe's own sub-language. A Web sandbox could allow "safe type annotations" in untrusted widgets. Note that such a pay-for-performance model is similar to Thorn's [92].

- We tried and failed to prove a basic non-interference property for ADsafe widgets. The property turned out to be false; it is violated in several ways by JavaScript's semantics and no Web sandbox can establish such a property. Developing and verifying that such a property holds is also an open problem.

[3]http://code.google.com/p/google-caja/wiki/Performance

Appendix A

Further Details of ADsafe Verification

Chapter 8 presents a type-based verification of ADsafe's original source code. The chapter documents the few refactorings necessary to satisfy the type-checker (section 8.7.1) and the bugs that we found (section 8.9). This chapter details the discrepancies between ADsafe's static checker, JSLint, and our type-checker, and details the few lines of code that remain unverified.

A.1 Differences Between JSLint and Typed Widgets

In section 8.5.2, we argue that if a widget passes JSLint, then it is also Widget-typable. However, Widget-typability does not imply that the widget passes JSLint. JSLint is a "code quality tool" that was retrofitted to perform security checks. Some of these code quality checks are irrelevant for safety, thus they reject safe code that the Widget-type admits. We give some examples below.

Banned Names The Widget-type states that `prototype`,`arguments`, and several other strings cannot be used as field names. However, it is safe to use them as identifiers:

```
ADSAFE.lib("Widget_", function() {
  var prototype, arguments;
});
```

However, JSLint rejects this program because it simply scans for "banned names".

Unused Identifiers JSLint requires identifiers to be used and hence rejects this program:

```
ADSAFE.lib("Widget_", function() {
  var unusedVariable;
});
```

However, this program is perfectly safe and Widget-typable.

Pascal-style Declarations JSLint requires all variable declarations to be at the head of a function, so it rejects this program:

```
ADSAFE.lib("test_", function () {
  var x = 34;
  x = x + 1;
  var y = 45;
});
```

However, this program is Widget-typable.

Appendix B

Characteristic Uses of Objects

This appendix translates the λ_S^{ob} examples in section 5.2 to various actual scripting languages.

Example 1

Prototype inheritance:

```
let Rect = { "area": func(self:?) . self["x"] * self["y"], "parent": null } in

let Cuboid = { "parent": Rect,

               "vol": func(self) . self["area"](self) * self["z"] } in

let shape = { "x": 2, "y": 5, "z": 10: "parent": Cuboid } in

let vol = shape["vol"](shape) // vol is 100
```

JavaScript Here is the equivalent program in JavaScript. Note that the `this` argument is implicit:

```
var Rect =   { area: function() { return this.x * this.y; },
               __proto__: null };
var Cuboid = { __proto__: Rect,
               vol: function() { return this.area() * this.z; } };
var shape = { x: 2, y: 5, z: 10, __proto__: Cuboid };
var vol = shape.vol(); // vol is 100
var f = vol;
var vol = f(); // ERROR: this.area is undefined
```

Lua This program can be written in Lua using *metatables*, which allow assigning a `parent`-like field:

```
Rect =   { area = function(self) return self.x * self.y end }
Cuboid = { vol = function(self) return self.area(self) * self.z end }
```

```
setmetatable(Cuboid, {__index = Rect})
shape = { x=2, y=5, z=10 }
setmetatable(shape, {__index = Cuboid})
vol = shape.vol(shape)
f = shape.vol
vol2 = f() // ERROR: attempt to index local self (a nil value)
```

Example 2

Extracting methods:

```
let ArrParent = { "slice": func(self:?,begin:?,end:?). ··· , ··· } in
let arr1 = { "0": 3, "1": 20, "2": 59, "length": 3, "parent": ArrParent }
let nodeList = { "0": htmlElementA, "1": htmlElementB, "2": htmlElementC,
               "len": 3, "parent": HTMLNodeListParent } in
let eltArray = ArrParent["slice"](nodeList,0,1)
// returns an array containing htmlElementA and htmlElementB
```

JavaScript This version of slice is built-in. We use DOM-manipulation functions to fetch array-like objects.

```
var ArrParent = Array.prototype;
var arr1 = [3, 20, 59]; // JavaScript desugars to an Array object
// Get all the links on a page:
var nodeList = document.getElementsByTagName("a");
// Using .call on a function allows us to provide the this arg
var eltArray = ArrParent.slice.call(nodeList, 0, 1)
// eltArray contains the first two elements in the list
```

Lua Lua objects allow trivial method extraction—Lua has similar array behavior to JavaScript as well, and any number-indexed dictionary can be used by library methods.

```
arr = {123, 45, 6}
table.sort(arr)
-- arr is now {1 = 6, 2 = 45, 3 = 123}
not_arr = {foo = "bar"}
not_arr[1] = 6
not_arr[2] = 5
not_arr[3] = 4
table.sort(not_arr)
-- not_arr is now {1 = 4, 2 = 5, 3 = 6, foo = "bar"}
```

Example 3

Bound methods:

```
let Rect = { "area": func(self:?) . self["x"] * self["y"], "parent": null } in
let Cuboid = { "parent": Rect,
              "vol": func(self) . self["area"](self) * self["z"] } in
let rec shape2 = {
  "x": 2, "y": 5, "z": 10",
  "_class_": Cuboid,
  "parent": {
    "vol": func() . shape2["_class_"]["vol"](shape2)
  }
} in
let f = shape2["vol"]
let vol2 = f() // vol2 is still 100, f closes over shape2
```

Python In Python, we can see this effect with classes:

```
class Rect(object):
  def area(self): return self.x * self.y
class Cuboid(Rect):
  def vol(self): return self.area() * self.z
shape2 = Cuboid()
shape2.x = 2; shape2.y = 5; shape2.z = 10
f = shape2.vol
vol2 = f() # vol2 is 100
```

Ruby In Ruby, we use `obj.method(:methname)` to access the method, and `method.call` to invoke it:

```
class Rect
  def area; self.x * self.y; end
end
class Cuboid < Rect
  def vol; self.area() * self.z; end
end
shape2 = Cuboid.new
def shape2.x; 2; end
def shape2.y; 5; end
def shape2.z; 10; end
f = shape2.method(:vol)
vol2 = f.call() # vol2 is 100
```

Example 4

Ad hoc private fields:

```
let safeGetField = Λα <: ?.func(obj:?,fieldName:?,default:?).
  if (fieldName matches "_.*_") default
  else if (obj hasfield fieldName) obj[fieldName] else default in
safeGetField(?)({ "_private_": 42, "pub": 23, "parent": null },
              "_private_", 0) // returns 0
```

JavaScript In JavaScript, this check could be performed with a regex. For a real-world example, see `reject_name` in ADsafe.[1]

```
function safeGetField(obj, field, default) {
  if(/_(.*)_/.test(field))        return default
  else {
    if(obj.hasOwnProperty(field)) return obj[field];
    else                          return default;
  }
}
```

Python A similar check works in Python. Note that variations on this pattern are found in production code inside Django.[2]

```
def safeGetField(obj, field, default):
  rx = re.compile(r"_(.*)_")
  if rx.match(field) is not None: return default
  else:
    if hasattr(obj, field):       return getattr(obj, field)
    else:                         return default
```

Ruby Note that several variations on this pattern are found in production code inside Ruby on Rails.[3]

```
def safeGetField(obj, field, default)
  return default unless /_(.*)_/.match(field).nil?
  return default unless obj.respond_to?(field)
  return obj.send(field.intern)
end
```

Example 5

Safe dictionary lookup:

[1] https:/github.com/douglascrockford/ADsafe/blob/master/adsafe.js#L254
[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#L1725

```
let safeAssign = Λα <: ?.func(dict:?,word:Str,value:?).dict["w_" + word = value]
let safeLookup = Λα <: ?.func(dict:?,world:Str,default:?).
  let lookup = "w_" + word in
  if (dict hasfield lookup) dict[lookup]
  else default
```

JavaScript While JavaScript does not have type abstraction, implementation of the core functionality is trivial:

```
function safeAssign(dict, word, value) { dict["w_" + word] = value; }
function safeLookup(dict, word, default) {
  var lookup = "w_" + word;
  if(dict.hasOwnProperty(word)) return dict[lookup];
  return default;
}
```

Such an implementation is necessary in JavaScript when objects are used as dicitonaries, because of the presence of the `__proto__` field in major browsers.

Python Python and Ruby both support dictionary-like objects natively, and don't need to use this pattern.

Appendix C

Proofs: Flow Typing

C.1 Full Typing Relation

$$\begin{array}{rcl} ty_1(\texttt{typeof}) & = & \top \to \mathsf{Str} \\ ty_2(\texttt{===}) & = & \top \times \top \to \mathsf{Bool} \\ ty_2(\texttt{-}) & = & \mathsf{Num} \times \mathsf{Num} \to \mathsf{Bool} \end{array}$$

$$\frac{\Sigma; \Gamma \vdash e : S \cdots \qquad ty_n(op_n) = S \cdots \to T}{\Sigma; \Gamma \vdash op_n(e \cdots) : T} \quad \text{(T-PrimApp)}$$

$$\Sigma; \Gamma \vdash num : \mathsf{Num} \quad \text{(T-Num)}$$

$$\Sigma; \Gamma \vdash str : \mathsf{Str} \quad \text{(T-Str)}$$

$$\Sigma; \Gamma \vdash bool : \mathsf{Bool} \quad \text{(T-Bool)}$$

$$\Sigma; \Gamma \vdash \texttt{undefined} : \mathsf{Undef} \quad \text{(T-Undef)}$$

$$\frac{\Sigma(l) = T}{\Sigma; \Gamma \vdash l : T} \quad \text{(T-Loc)}$$

$$\frac{\Gamma(x) = T}{\Sigma; \Gamma \vdash x : T} \quad \text{(T-Id)}$$

$$\frac{\Sigma;\Gamma', x:S,\cdots \vdash e:T \qquad \Gamma' = \Gamma \text{ with labels removed}}{\Sigma;\Gamma \vdash \texttt{func(}x\cdots\texttt{)}\ \texttt{:}S\cdots\to T\texttt{\{}\ e\ \texttt{\}} : S\cdots\to T} \quad \text{(T-Abs)}$$

$$\frac{\Sigma;\Gamma \vdash e : S\cdots \qquad \Sigma;\Gamma \vdash f : S\cdots \to T}{\Sigma;\Gamma \vdash f\texttt{(}e\cdots\texttt{)} : T} \quad \text{(T-App)}$$

$$\frac{\Sigma;\Gamma \vdash e_1 : S \qquad \Sigma; x:S,\Gamma \vdash e_2 : T}{\Sigma;\Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ e_2 : T} \quad \text{(T-Let)}$$

$$\frac{\Sigma;\Gamma \vdash e : T}{\Sigma;\Gamma \vdash \texttt{ref}\ e : \mathsf{Ref}\ T} \quad \text{(T-Ref)}$$

$$\frac{\Sigma;\Gamma \vdash e : \mathsf{Ref}\ T}{\Sigma;\Gamma \vdash \texttt{deref}\ e : T} \quad \text{(T-Deref)}$$

$$\frac{\Sigma;\Gamma \vdash e_1 : \mathsf{Ref}\ S \qquad \Sigma;\Gamma \vdash e_2 : T \qquad T <: S}{\Sigma;\Gamma \vdash \texttt{setref}\ e_1\ e_2 : \mathsf{Ref}\ \ T} \quad \text{(T-SetRef)}$$

$$\frac{\Sigma;\Gamma \vdash e_1 : \mathsf{Bool} \qquad \Sigma;\Gamma \vdash e_2 : T \qquad \Sigma;\Gamma \vdash e_3 : T}{\Sigma;\Gamma \vdash \texttt{if (}e_1\texttt{) \{ }e_2\texttt{ \} else \{ }e_3\texttt{ \}} : T} \quad \text{(T-If)}$$

$$\frac{\Sigma;\Gamma, \mathit{label} : T \vdash e : T}{\Sigma;\Gamma \vdash \mathit{label}\texttt{:}T\ \texttt{\{}\ e\ \texttt{\}} : T} \quad \text{(T-Label)}$$

$$\frac{\Gamma(\mathit{label}) = T \qquad \Sigma,\Gamma \vdash e : T}{\Sigma;\Gamma \vdash \texttt{break}\ \mathit{label}\ e : \bot} \quad \text{(T-Break)}$$

$$\frac{\Sigma;\Gamma \vdash e : S \qquad S <: T}{\Sigma;\Gamma \vdash e : T} \quad \text{(T-Sub)}$$

$$\frac{\forall l \in \mathit{dom}(\Sigma).\Sigma;\cdot \vdash \sigma(l) : \Sigma(l) \qquad \mathit{dom}(\Sigma) = \mathit{dom}(\sigma)}{\Sigma \vdash \sigma} \quad \text{(T-}\sigma\text{)}$$

C.2 Type Safety

Lemma 10 (Progress) *If $\Sigma, \cdot \vdash e : T$ and $\Sigma \vdash \sigma$ then either:*

i. $e \in v$, or

ii. There exists σ', e', such that $\sigma e \to \sigma' e'$, or

iii. There exists an E, such that $e = E\langle \textsf{tagerr} \rangle$.

Proof: By lemma 12, cases (i.) and (iii.) are immediate, leaving only the the case where $e = E\langle ae \rangle$. By lemma 13, there exist S and Γ, such that $\Sigma; \Gamma \vdash ae : S$. By case-analysis on ae, applying inversion (lemma 17) and canonical forms (lemma 18) where indicated:

- Case $ae = \textsf{ref}\ v$. By E-Ref, $\sigma E\langle \textsf{ref}\ e \rangle \to \sigma, (l, v)\langle l \rangle$ where $l \notin dom(\sigma)$.
- Case $ae = \textsf{deref}\ v$. By inversion, $\Sigma; \Gamma \vdash v : \textsf{Ref}\ \ S'$. By canonical forms, $v = l$ and $l \in dom(\Sigma)$. By T-σ, $l \in dom(\sigma)$. By E-Deref, $\sigma E\langle \textsf{deref}\ \textsf{l} \rangle \to \sigma E\langle \sigma(l) \rangle$.
- Case $ae = \textsf{setref}\ v_1\ v_2$. By inversion, $\Sigma; \Gamma \vdash v_1 : \textsf{Ref}\ \ S_1$. By canonical forms, $v_1 = l$, hence $l \in dom(\sigma)$. By E-SetRef, $\sigma E\langle \textsf{setref}\ l\ v_2 \rangle \to \sigma[l := v_2]E\langle l \rangle$.
- Case $ae = label\texttt{:}T\texttt{\{}\ E_1\langle \textsf{break}\ label\ v \rangle\ \texttt{\}}$. By E-Break, $\sigma E\langle label\texttt{:}T\texttt{\{}\ E_1\langle \textsf{break}\ label\ v \rangle\ \texttt{\}} \rangle \to \sigma E\langle v \rangle$.
- Case $ae = label\texttt{:}T\ \texttt{\{}\ v\ \texttt{\}}$. By E-Label-Pop, $\sigma E\langle label\texttt{:}T\ \texttt{\{}\ v\ \texttt{\}} \rangle \to \sigma E\langle v \rangle$.
- Case $ae = \textsf{let}\ x\ \texttt{=}\ v\ \textsf{in}\ e$. By E-Let, $\sigma E\langle \textsf{let}\ x\ \texttt{=}\ v\ \textsf{in}\ e \rangle \to \sigma E\langle e[x/v] \rangle$.
- Case $ae = v_f\texttt{(}v \cdots\texttt{)}$. By canonical forms, $\Sigma; \Gamma \vdash v_f : S' \cdots \to S$. By inversion (lemma 17), $v_f = \textsf{func}\texttt{(}x \cdots\texttt{)}\ \texttt{\{}\ e\ \texttt{\}}$. By E-Abs, $\sigma E\langle \textsf{func}\texttt{(}x \cdots\texttt{)}\ \texttt{\{}\ e\ \texttt{\}(}v \cdots\texttt{)} \rangle \to \sigma E\langle e'[x/v] \cdots \rangle$.
- Case $ae = op_n\texttt{(}v \cdots\texttt{)}$ is similar to the case above.
- Case $ae = \textsf{tagcheck}\ R\ v$. If $\delta_1(\textsf{typeof}, v) \in R$, then $\sigma E\langle \textsf{tagcheck}\ R\ v \rangle \to \sigma E\langle v \rangle$ by E-TagCheck. If not, then $\sigma E\langle \textsf{tagcheck}\ R\ v \to \sigma E\langle \textsf{tagerr} \rangle$ by E-TagCheck-Err.

■

Lemma 11 (Preservation) *If $\Sigma_1; \cdot \vdash e_1 : T$, $\Sigma_1 \vdash \sigma_1$, and $\sigma_1 e_1 \to \sigma_2 e_2$, then there exists a Σ_2, such that:*

i. $\Sigma_2; \cdot \vdash e_2 : T$*,*

ii. $\Sigma_2 \vdash \sigma_2$*, and*

iii. $\Sigma_1 \subseteq \Sigma_2$*.*

Proof: By case-analysis of the reduction rules, there exists an evaluation context (E), an active expression (ae), and an expression (e'), such that $e_1 = E\langle ae\rangle$ and $e_2 = E\langle e'\rangle$. Hence,

$$\sigma_1 E\langle ae\rangle \rightarrow \sigma_2 E\langle e'\rangle$$

By lemma 13, there exist Γ and S, such that $\Sigma_1; \Gamma \vdash ae : S$ and Γ only contains labels. We will prove that $\Sigma_2; \Gamma \vdash e' : S$, so that $\Sigma_2; \Gamma \vdash E\langle e'\rangle : T$.

We proceed by case-analysis on ae, using inversion (lemma 17) where specified:

- $\textsf{\textbf{let}}\ x\ \texttt{=}\ v\ \textsf{in}\ e$. By inversion, $\Sigma_1; \Gamma \vdash v : U$ and $\Sigma_1; x : U, \Gamma \vdash e : S$. By E-Let, $ae \rightarrow e[x/v]$. By substitution (lemma 16), $\Sigma; \Gamma \vdash e[x/v] : S$.

- $\texttt{(}\textsf{\textbf{func}}\ \texttt{(}x\cdots\texttt{)}\ \texttt{\{}\ e\ \texttt{\})(}v\cdots\texttt{)}$. By inversion and canonical forms, $\Sigma_1; \Gamma \vdash v : U \cdots$ and $\Sigma_1; x : U \cdots \vdash e : S$. By E-App, $\sigma\texttt{(}\textsf{\textbf{func}}\ \texttt{(}x\cdots\texttt{)}\ \texttt{\{}\ e\ \texttt{\})(}v\cdots\texttt{)} \rightarrow \sigma e[x/v]\cdots$. By substitution (lemma 16), $\Sigma_1; \cdot \vdash e[x/v]\cdots : S$.

- $\mathit{label}\texttt{:}S'\ \texttt{\{}\ v\ \texttt{\}}$. By inversion, $\Sigma_1; \mathit{label} : S', \Gamma \vdash v : S'$ and $S' <: S$. By E-Label-Pop, $\sigma \mathit{label}\texttt{:}S\texttt{\{}\ v\ \texttt{\}} \rightarrow \sigma v$. Typing values does not require labels in the environment (lemma 15), hence $\Sigma_1; \Gamma \vdash v : S'$. The conclusion follows by T-Sub.

- $\mathit{label}\texttt{:}S'\ \texttt{\{}\ E'\langle\textsf{\textbf{break}}\ \mathit{label}\ v\rangle\texttt{\}}$. By inversion, $\Sigma_1; \mathit{label} : S', \Gamma \vdash E'\langle\textsf{\textbf{break}}\ \mathit{label}\ v\rangle : S'$ and $S' <: S$. By E-Break, $\sigma E'\langle\textsf{\textbf{break}}\ \mathit{label}\ v\rangle \rightarrow \sigma v$. Since the evaluation context, E' cannot bind identifiers and values can be typed without labels (lemma 15), $\Sigma_1; \Gamma \vdash v : S'$.

- $\textsf{\textbf{ref}}\ v$. By inversion, $\Sigma_1; \Gamma \vdash v : U$ and $\textsf{Ref}\ \ U <: S$. By E-Ref, for an $l \notin \mathit{dom}(\sigma_1)$, $\sigma_2 = \sigma_1, (l, v)$ and $\sigma_1\textsf{\textbf{ref}}\ v \rightarrow \sigma_2 l$. Let $\Sigma_2 = l : U, \Sigma_1$. By T-Loc, $\Sigma_2; \cdot \vdash l : \textsf{Ref}\ \ U$, followed by T-Sub with $\textsf{Ref}\ \ U <: S$.

- $\textsf{\textbf{deref}}\ l$. By inversion, $\Sigma_1; \Gamma \vdash l : \textsf{Ref}\ \ U$ with $U <: S$. By hypothesis and T-Loc, $\Sigma_1(l) = U$. By $\Sigma_1 \vdash \sigma_1$, there exists a v such that $\sigma_1(l) = v$ and $\Sigma; \Gamma \vdash v : U$. By E-Deref, $\sigma_1\textsf{\textbf{deref}}\ l \rightarrow \sigma_1 v$. By T-Sub, $\Sigma_1; \Gamma \vdash v : S$.

- $\textsf{setref}\ l\ v$. By inversion, $\Sigma_1;\Gamma \vdash l : \textsf{Ref}\ \ S'$, $\Sigma_1;\Gamma \vdash v : U$, $U <: S'$, and $\textsf{Ref}\ \ S' <: S$. By T-Loc, $\Sigma_1(l) = S'$. By E-SetRef, $\sigma_1\textsf{setref}\ l\ v \to \sigma_1[l/v]l$. Since Γ does not bind identifiers and by lemma 15, $\Sigma_1;\cdot \vdash v : U$. By T-Sub, $\Sigma;\cdot \vdash v : S'$. Hence, $\Sigma_1 \vdash \sigma_1[l/v]$.

- $\textsf{tagcheck}\ R\ v$.

  - $\sigma_1 E\langle\textsf{tagcheck}\ R\ v\rangle \to \sigma_1 E\langle v\rangle$, and $\delta_1(\textbf{typeof}, v) \in R$. By inversion, $\Sigma_1;\Gamma \vdash v : U$ and $S' = \mathit{static}(R, U)$ for $S' <: S$. By definition of static, $S' <: U$. By case-analysis of v:

    * $v = \textbf{func}\ \texttt{(}x\cdots\texttt{)}\ \texttt{:}T_a\cdots\to T_r\texttt{\{}\ e\ \texttt{\}}$, hence $\Sigma_1;\cdot \vdash v : T_a\cdots\to T_r$ and $T_a\cdots\to T_r <: U$. By lemma 19, $\mathit{static}(R, T_a\cdots\to T_r) <: \mathit{static}(R, U) <: S$. By lemma 20, $\textsf{"function"} \in \mathit{runtime}(U)$. Hence, $\mathit{static}(R,, T_a\cdots\to T_r) = T_a\cdots\to T_r$.

    * Constants are symmetric to functions.

  - $\sigma_1 E\langle\textsf{tagcheck}\ R\ v\rangle \to \sigma_1 E\langle\textsf{tagerr}\rangle$. Apply T-TagErr and T-Sub.

∎

Definition 1 (Active Expressions) *An active expression, ae is one of:*

$$
\begin{array}{rcl}
ae & = & \textbf{let}\ x\ \texttt{=}\ v\ \textit{in}\ e \\
 & | & v_f\ \texttt{(}v\cdots\texttt{)} \\
 & | & op_n\ \texttt{(}v\cdots\texttt{)} \\
 & | & label\texttt{:}T\ \texttt{\{}\ v\ \texttt{\}} \\
 & | & \textbf{ref}\ v \\
 & | & \textbf{deref}\ v \\
 & | & \textsf{setref}\ v_1\ v_2 \\
 & | & label\texttt{:}T\texttt{\{}\ E\langle\textbf{break}\ v\rangle\ \texttt{\}}\ \textit{where label} \notin E \\
 & | & \textsf{tagcheck}\ R\ v
\end{array}
$$

Lemma 12 *For all closed expressions e, either:*

i. $e \in v$,

ii. *there exist e and ae such that* $e = E\langle ae\rangle$, *or*

iii. $e = E\langle\textsf{tagerr}\rangle$, *for some* E.

Proof: By definition of e, v, and E. ■

Lemma 13 (Closed Active Expressions) *For all* Σ, T, E, ae*, if* $\Sigma; \cdot \vdash E\langle ae\rangle : T$*, then there exist* S *and* Γ*, where* Γ *only contains labels, such that* $\Sigma; \Gamma \vdash ae : S$*.*

Proof: there exists a subdeduction of the typing derivation, such that $\Sigma; \Gamma \vdash ae : S$. Suppose an identifier, $x \in dom(\Gamma)$. Then, ae is in a program context $C\langle \texttt{let}\ x\ \texttt{=}\ e_1\ \texttt{in}\ \bullet\rangle$ or $C\langle \texttt{func}(x\cdots)\ \texttt{\{}\ \bullet\ \texttt{\}}\rangle$. However, this cannot occur since ae is in an evaluation context. ■

Lemma 14 *For all* E*, if* $\sigma_1 e_1 \to \sigma_2 e_2$*, then* $\sigma_1 E\langle e_1\rangle \to \sigma_2 E\langle e_2\rangle$*.*

Proof: by case-analysis of the reduction rules, there exist E', e_1', e_2', such that $e_1 = E'\langle e_1'\rangle$, $e_2 = E'\langle e_2'\rangle$, and $\sigma_1 E'\langle e_1'\rangle \to \sigma_2 E'\langle e_2'\rangle$. By the same reduction rule, $\sigma_1 E\langle E'\langle e_1'\rangle\rangle \to \sigma_2 E\langle E'\langle e_2'\rangle\rangle$. ■

Lemma 15 *If* $\Sigma; \Gamma \vdash v : T$ *then* $\Sigma; \Gamma' \vdash v : T$*, where* $\Gamma' = \Gamma$ *with labels removed.*

Proof: the only interesting case is $v = \texttt{func}(x\cdots)\ \texttt{\{}\ e\ \texttt{\}}$, where e is typed in an extension of Γ with labels removed, i.e., Γ'. ■

Lemma 16 (Substitution) *If* $\Sigma; x : S, \Gamma \vdash e : T$ *and* $\Sigma; \Gamma \vdash v : S$*, then* $\Sigma; \Gamma \vdash e[x/v] : T$*.*

Proof: by induction on the typing derivation. ■

Lemma 17 (Inversion) *If:*

- $\Sigma; \Gamma \vdash \texttt{ref}\ e : T$*, then* $\mathit{Ref}\ \ S <: T$ *and* $\Sigma; \Gamma \vdash e : S$*,*
- $\Sigma; \Gamma \vdash \texttt{deref}\ e : T$*, then* $\Sigma; \Gamma \vdash e : \mathit{Ref}\ \ S$ *with* $S <: T$*,*
- $\Sigma; \Gamma \vdash \mathit{setref}\ e_1\ \ e_2 : T$*, then* $\Sigma; \Gamma \vdash e_1 : \mathit{Ref}\ \ S$*,* $\Sigma; \Gamma \vdash e_2 : U$*,* $U <: S$*, and* $\mathit{Ref}\ \ S <: T$*,*
- $\Sigma; \Gamma \vdash e_f\,(e\cdots) : T$*, then* $\Sigma; \Gamma \vdash e_f : S\cdots \to T'$*,* $\Sigma; \Gamma \vdash e : S\cdots$*, and* $T' <: T$*.*
- $\Sigma; \Gamma \vdash \mathit{label}{:}S\ \texttt{\{}\ e\ \texttt{\}} : T$*, then* $S <: T$ *and* $\sigma; \mathit{label} : S, \Gamma \vdash e : S$*.*
- $\Sigma; \Gamma \vdash \texttt{let}\ x\ \texttt{=}\ e_1\ \mathit{in}\ e_2 : T$*, then* $\Sigma; \Gamma \vdash e_1 : S$ *and* $\Sigma; x : S, \Gamma \vdash e_2 : T$*.*
- $\Sigma; \Gamma \vdash \mathit{tagcheck}\ R\ \ e : T$*, then* $\Sigma; \Gamma \vdash e = S$ *and* $\mathit{static}(R, S) = T'$*, where* $T' <: T$*.*

Proof: by induction on the typing derivation. In all cases, only T-Sub and one other typing rule apply. T-Sub requires induction and the other latter is immediate.

- Only T-Ref and T-Sub apply. T-Ref is immediate with $\mathsf{Ref}\ \ S = T$. For T-Sub, by inversion, $\Sigma; \Gamma \vdash \textsf{\textbf{ref}}\ \mathsf{e} : T'$ and $T' <: T$. By induction, $T' = \mathsf{Ref}\ \ S$ and $\Sigma; \Gamma \vdash e : S$.
- Only T-Deref and T-Sub apply. T-Deref is immediate with $T <: T$. For case T-Sub, by inversion $\Sigma; \Gamma \vdash \textsf{\textbf{deref}}\ \mathsf{e} : S$ and $S <: T$. By induction, $\Sigma; \Gamma \vdash e : \mathsf{Ref}\ \ S'$ with $S' <: S$. By S-Trans, $S' <: T$
- Only T-SetRef and T-Sub apply. T-SetRef is immediate, with $\mathsf{Ref}\ \ S = T$. For T-Sub, by inversion, $\Sigma; \Gamma \vdash \mathsf{setref}\ e_1\ \ e_2 : T'$ and $T' <: T$. By induction, $\Sigma; \Gamma \vdash e_1 : \mathsf{Ref}\ \ S$, $\Sigma; \Gamma \vdash e_2 : U$, $U <: S$, and $\mathsf{Ref}\ \ S <: T'$. By S-Trans, $\mathsf{Ref}\ \ S <: T$.
- Only cases T-App and T-Sub apply. T-App is immediate. For T-Sub, the conclusion follows by induction and S-Arr.
- Only cases T-Sub and T-Label apply. T-Label follows immediately with $S = T$ and S-Refl. For T-Sub, $\Sigma; \Gamma \vdash \mathit{label}{:}S\ \texttt{\{}\ e\ \texttt{\}} : T'$ and $T' <: T$. By induction, $\Sigma; \mathit{label} : S, \Gamma \vdash e : S$ and $S <: T'$. Hence, by S-Trans, $S <: T$.
- Only T-Let and T-Sub apply. T-Let is immediate. For T-Sub, by induction, $\Sigma; \Gamma \vdash e_1 : S$, $\Sigma; x : S, \Gamma \vdash e_2 : T'$, and $T' <: T$. Apply T-Sub.
- Only T-TagCheck and T-Sub apply. T-TagCheck is immediate. For T-Sub, by induction, it holds for a $T' <: T$.

■

Lemma 18 (Canonical Forms) *For $\Sigma; \Gamma \vdash v : T$:*

i. If $T = \mathsf{Ref}\ \ S$ then $v = \mathit{loc}$ and $\Sigma(\mathit{loc}) <: S$.

ii. If $T = U \cdots \to S$ then $v = \textsf{\textbf{func}}\,(x \cdots)\ \texttt{\{}\ e\ \texttt{\}}$.

Proof: By induction on the typing derivation.

i. Only T-Loc and T-Sub apply. T-Loc is immediate. For T-Sub, $\Sigma; \Gamma \vdash v : T'$ with $T' <: T$. By induction, $v = \mathit{loc}$, $T' = \mathsf{Ref}\ \ S'$, and $\Sigma(\mathit{loc}) <: S'$.

ii. Only T-Abs and T-Sub apply. T-Abs is immediate, and T-Sub follows by induction.

■

Lemma 19 (*static* commutes with subtyping) *If $S <: T$, then $static(R, S) <: static(R, T)$.*

Proof: by induction on $S <: T$. ■

Lemma 20 *If $\Sigma; \Gamma \vdash v : T$, then $\delta_1(\textsf{typeof}, v) \in runtime(T)$.*

Proof: by case analysis of v. ■

C.3 CPS Transformation

The following four mutually-recursive functions define our CPS transformation: $\mathcal{C}_k$ transforms expressions, Φ transforms values, $\mathcal{K}_k$ transforms evaluation contexts, and $\mathcal{P}_k$ transforms program states (expressions with stores). This style of transformation is due to Sabry and Felleisen [75]. Our presentation has a few minor distinctions:

- The semantics of our source language directly specifies the evaluation of **break**, which is a very restricted form of *call/cc*. Hence, **break** is an expression transformed by $\mathcal{C}_k$, instead of a value transformed by Φ.
- We use a global store instead of a syntactic store, hence the need for $\mathcal{P}_k$.
- We use **let**-expressions to avoid continuation-passing operators.
- We do not employ a fully-compacting transformation. For example, we do not have a special-case for $\mathcal{C}_k[\![E\langle\texttt{(}\textbf{func}\texttt{(}x\cdots\texttt{):T \{ }e\texttt{ \})(}v\cdots\texttt{)}\rangle]\!]$, as it would introduce interprocedural flows in the source language. (In contrast, see [75, Definition 5].)

$\boxed{\mathcal{C}_k : e \to M}$

$$\begin{aligned}
\mathcal{C}_k[\![v]\!] &= k(\Phi[\![v]\!]) \\
\mathcal{C}_k[\![E\langle \textbf{let}\ \texttt{(}x\ \texttt{=}\ v\texttt{)}\ e\rangle]\!] &= \textbf{let}\ \texttt{(}x\ \texttt{=}\ \Phi[\![v]\!]\texttt{)}\mathcal{C}_k[\![E\langle e\rangle]\!] \\
\mathcal{C}_k[\![E\langle v_f\texttt{(}v\cdots\texttt{)}\rangle]\!] &= \Phi[\![v_f]\!]\texttt{(}\mathcal{K}_k[\![E]\!]\texttt{,}\Phi[\![v]\!]\cdots\texttt{)} \\
\mathcal{C}_k[\![E\langle op_n\texttt{(}v\cdots\texttt{)}\rangle]\!] &= \textbf{let}\ x\ \texttt{=}\ op_n\texttt{(}\Phi[\![v]\!]\cdots\texttt{)}\ \textsf{in}\ \mathcal{K}_k[\![E]\!]\texttt{(}x\texttt{)} \\
\mathcal{C}_k[\![E\langle \textbf{if}\ \texttt{(}v\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ e_3\ \texttt{\}}\rangle]\!] &= \texttt{(}\textbf{func}\ \texttt{(}k\texttt{)}\ \texttt{\{}\ \textbf{if}\ \texttt{(}\Phi[\![v]\!]\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![e_2]\!]\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ \mathcal{C}_k[\![e_3]\!]\ \texttt{\}}\ \texttt{\})(}\mathcal{K}_k[\![E]\!]\texttt{)} \\
\mathcal{C}_k[\![E_1\langle label\texttt{:}\ \texttt{\{}\ E_2\langle \textbf{break}\ label\ v\rangle\ \texttt{\}}]\!] &= \mathcal{K}_k[\![E_1]\!]\texttt{(}\Phi[\![v]\!]\texttt{)},\ \text{when}\ label \notin E_2 \\
\mathcal{C}_k[\![E\langle label\texttt{:\{}\ v\ \texttt{\}}\rangle]\!] &= \mathcal{C}_k[\![E\langle v\rangle]\!] \\
\mathcal{C}_k[\![E\langle \textbf{ref}\ v\rangle]\!] &= \textbf{let}\ x\ \texttt{=}\ \textbf{ref}\ \Phi[\![v]\!]\ \textsf{in}\ \mathcal{K}_k[\![E]\!]\texttt{(}x\texttt{)} \\
\mathcal{C}_k[\![E\langle \textbf{deref}\ v\rangle]\!] &= \textbf{let}\ x\ \texttt{=}\ \textbf{deref}\ \Phi[\![v]\!]\ \textsf{in}\ \mathcal{K}_k[\![E]\!]\texttt{(}x\texttt{)} \\
\mathcal{C}_k[\![E\langle \textsf{setref}\ l\ v\rangle]\!] &= \textbf{let}\ x\ \texttt{=}\ \textsf{setref}\ l\ \Phi[\![v]\!]\ \textsf{in}\ \mathcal{K}_k[\![E]\!]\texttt{(}l\texttt{)} \\
\mathcal{C}_k[\![E\langle \textsf{tagcheck}\ R\ v\rangle]\!] &= \textbf{let}\ x\ \texttt{=}\ \textsf{tagcheck}\ R\ \Phi[\![v]\!]\ \textsf{in}\ \mathcal{K}_k[\![E]\!]\texttt{(}x\texttt{)}
\end{aligned}$$

$\boxed{\Phi : v \to V}$

$$\begin{aligned}
\Phi[\![x]\!] &= x \\
\Phi[\![\textbf{func}\texttt{(}x\cdots\texttt{):}S\cdots\to T\ \texttt{\{}\ e\ \texttt{\}}]\!] &= \textbf{func}\texttt{(}k\texttt{,}x\cdots\texttt{):}(T\to\bot)\times S\cdots\to\bot\ \texttt{\{}\ \mathcal{C}_k[\![e]\!]\ \texttt{\}} \\
\Phi[\![c]\!] &= c \\
\Phi[\![l]\!] &= l
\end{aligned}$$

$\boxed{\mathcal{K}_k : E \rightarrow V}$

$$
\begin{array}{rcl}
\mathcal{K}_k[\![\bullet]\!] & = & k \\
\mathcal{K}_k[\![E\langle \textbf{let}\ \texttt{(}x\ \texttt{=}\ \bullet\texttt{)}\ e\rangle]\!] & = & \textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle e\rangle]\!]\ \texttt{\}} \\
\mathcal{K}_k[\![E\langle \bullet\texttt{(}e\cdots\texttt{)}\rangle]\!] & = & \textbf{func}\texttt{(}f\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle f\texttt{(}e\cdots\texttt{)}\rangle]\!]\ \texttt{\}} \\
\mathcal{K}_k[\![E\langle v_f\texttt{(}v\cdots\bullet\, e\cdots\texttt{)}\rangle]\!] & = & \textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle v_f\texttt{(}v\cdots xe\cdots\texttt{)}\rangle]\!]\ \texttt{\}} \\
\mathcal{K}_k[\![E\langle op_n\texttt{(}v\cdots\bullet\, e\cdots\texttt{)}\rangle]\!] & = & \textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle op_n\texttt{(}v\cdots xe\cdots\texttt{)}\rangle]\!]\ \texttt{\}} \\
\mathcal{K}_k[\![E\langle \textbf{if}\ \texttt{(}\bullet\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ e_3\ \texttt{\}}\rangle]\!] & = & \textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle \textbf{if}\ \texttt{(}x\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ e_3\ \texttt{\}}\rangle]\!]\ \texttt{\}} \\
\mathcal{K}_k[\![E\langle \textbf{ref}\ \bullet\rangle]\!] & = & \textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle \textbf{ref}\ x\rangle]\!]\ \texttt{\}} \\
\mathcal{K}_k[\![E\langle \textbf{deref}\ \bullet\rangle]\!] & = & \textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle \textbf{deref}\ x\rangle]\!]\ \texttt{\}} \\
\mathcal{K}_k[\![E\langle \textsf{setref}\ \bullet\ e\rangle]\!] & = & \textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle x\ \texttt{=}\ e\rangle]\!]\ \texttt{\}} \\
\mathcal{K}_k[\![E\langle \textsf{setref}\ v\ \bullet\rangle]\!] & = & \textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle v\ \texttt{=}\ x\rangle]\!]\ \texttt{\}} \\
\mathcal{K}_k[\![E\langle \textbf{break}\ label\ \bullet\rangle]\!] & = & \textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle \textbf{break}\ label\ x\rangle]\!]\ \texttt{\}} \\
\mathcal{K}_k[\![E\langle label\texttt{:}\ \texttt{\{}\ \bullet\ \texttt{\}}\rangle]\!] & = & \textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle label\texttt{:}\ \texttt{\{}\ x\ \texttt{\}}\ ]\!]\ \texttt{\}} \\
\mathcal{K}_k[\![E\langle \textsf{tagcheck}\ R\ \bullet\rangle]\!] & = & \textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle \textsf{tagcheck}\ R\ x\rangle]\!]\texttt{\}}
\end{array}
$$

$\boxed{\mathcal{P}_k : \sigma e \rightarrow SM}$

$$\mathcal{P}_l[\![(l,v)\cdots e]\!] \quad = \quad (l,\Phi(v))\cdots\mathcal{C}_k[\![e]\!]$$

Lemma 21 (Soundness of $\mathcal{P}_k$) *If $\sigma e \rightarrow_R \sigma' e'$, then $\mathcal{P}_k[\![\sigma e]\!] \twoheadrightarrow_{\{R,E\text{-}Let,\widehat{\beta}\}} \mathcal{P}_k[\![\sigma' e']\!]$.*

Proof: The proof is by induction on the size of σe and proceeds by case-analysis of the definition of $\rightarrow$. To prove that only R, E-Let, and $\widehat{\beta}$ are applied, we simply examine the reduction sequences below. Note that the auxilliary lemma 22 only employs $\widehat{\beta}$.

- Case E-Let.

$$
\begin{array}{cll}
 & S\ \mathcal{C}_k[\![E\langle \textbf{let}\ \texttt{(}x\ \texttt{=}\ v\texttt{)}\ e\rangle]\!] & \\
= & S\ \textbf{let}\ \texttt{(}x\ \texttt{=}\ \Phi[\![v]\!]\texttt{)}\ \mathcal{C}_k[\![E\langle e\rangle]\!] & \\
\rightarrow & S\ \mathcal{C}_k[\![E\langle e\rangle]\!][x/\Phi[\![v]\!]] & \text{E-Let} \\
= & S\ \mathcal{C}_k[\![E\langle e\rangle[x/v]]\!] & \text{lemma 23}
\end{array}
$$

which is $\mathcal{C}_k$, applied to the the RHS of E-Let.

- Case E-Break.

$$\begin{array}{lll} & S\ \mathcal{C}_k[\![E_1\langle \mathit{label}\texttt{:}T\ \texttt{\{}\ E_2\langle \textbf{break}\ \mathit{label}\ v\rangle\ \texttt{\}}\rangle]\!] & \\ = & S\ \mathcal{K}_k[\![E_1]\!]\texttt{(}\Phi[\![v]\!]\texttt{)} & \\ \twoheadrightarrow & S\ \mathcal{C}_k[\![E_1\langle v\rangle]\!] & \text{lemma 22} \end{array}$$

- Case E-Label-Pop.

$$\begin{array}{ll} & S\ \mathcal{C}_k[\![E\langle \mathit{label}\texttt{:}T\ \texttt{\{}\ v\ \texttt{\}}\rangle]\!] \\ = & S\ \mathcal{C}_k[\![E\langle v\rangle]\!] \end{array}$$

- Case E-Ref.

$$\begin{array}{lll} & S\ \mathcal{C}_k[\![E\langle \textbf{ref}\ v\rangle]\!] & \\ = & S\ \textbf{let}\ x\ \texttt{=}\ \textbf{ref}\ \Phi[\![v]\!]\ \textsf{in}\ \mathcal{K}_k[\![E]\!]\texttt{(}x\texttt{)} & \\ \rightarrow & S,(l,\Phi[\![v]\!])\ \textbf{let}\ x\ \texttt{=}\ l\ \textsf{in}\ \mathcal{K}_k[\![E]\!]\texttt{(}x\texttt{)} & \text{E-Ref} \\ \rightarrow & S,(l,\Phi[\![v]\!])\ \mathcal{K}_k[\![E]\!]\texttt{(}l\texttt{)} & \text{E-Let} \\ = & S,(l,\Phi[\![v]\!])\ \mathcal{K}_k[\![E]\!]\texttt{(}\Phi(l)\texttt{)} & \\ \twoheadrightarrow & S,(l,\Phi[\![v]\!])\ \mathcal{C}_k[\![E\langle l\rangle]\!] & \text{lemma 22} \end{array}$$

- Case E-Deref.

$$\begin{array}{lll} & S\ \mathcal{C}_k[\![E\langle \textbf{deref}\ l\rangle]\!] & \\ = & S\ \textbf{let}\ x\ \texttt{=}\ \textbf{deref}\ l\ \textsf{in}\ \mathcal{K}_k[\![E]\!]\texttt{(}x\texttt{)} & \\ \rightarrow & S\ \textbf{let}\ x\ \texttt{=}\ S(l)\ \textsf{in}\ \mathcal{K}_k[\![E]\!]\texttt{(}x\texttt{)} & \text{E-Deref and } dom(S) = dom(\sigma) \\ \rightarrow & S\ \mathcal{K}_k[\![E]\!]\texttt{(}S(l)\texttt{)} & \text{E-Let} \\ \twoheadrightarrow & S\ \mathcal{C}_k[\![E\langle v\rangle]\!] & \text{lemma 22, where } S(l) = \Phi(v) \end{array}$$

- Case E-SetRef.

$$\begin{array}{lll} & S\ \mathcal{C}_k[\![E\langle \textsf{setref}\ l\ v\rangle]\!] & \\ = & S\ \textbf{let}\ x\ \texttt{=}\ \textsf{setref}\ l\ \Phi[\![v]\!]\ \textsf{in}\ \mathcal{K}_k[\![E]\!]\texttt{(}l\texttt{)} & \\ \rightarrow & (l,\Phi[\![v]\!])S\ \textbf{let}\ x\ \texttt{=}\ l\ \textsf{in}\ \mathcal{K}_k[\![E]\!]\texttt{(}l\texttt{)} & \\ \rightarrow & (l,\Phi[\![v]\!])S\ \mathcal{K}_k[\![E]\!]\texttt{(}l\texttt{)} & \text{since } x \text{ is unused} \\ \twoheadrightarrow & (l,\Phi[\![v]\!])S\ \mathcal{C}_k[\![E\langle l\rangle]\!] & \text{by lemma 22, since } \phi(l) = l \end{array}$$

- Case E-TagCheck.

$$\begin{array}{lll} & S\ \mathcal{C}_k[\![E\langle \textsf{tagcheck}\ R\ v\rangle]\!] & \\ = & S\ \textsf{\textbf{let}}\ x\ \texttt{=}\ \textsf{tagcheck}\ R\ \Phi[\![v]\!]\ \textsf{in}\ \mathcal{K}_k[\![E]\!]\texttt{(}x\texttt{)} & \\ \rightarrow & S\ \mathcal{K}_k[\![E]\!]\texttt{(}\Phi[\![v]\!]\texttt{)} & \text{since } \textsf{\textbf{typeof}}\texttt{(}v\texttt{)} = \textsf{\textbf{typeof}}\texttt{(}\Phi[\![v]\!]\texttt{)} \\ \twoheadrightarrow & S\ \mathcal{C}_k[\![E\langle v\rangle]\!] & \text{lemma 22} \end{array}$$

■

Lemma 22 $S\mathcal{K}_k[\![E]\!]\texttt{(}\Phi[\![v]\!]\texttt{)} \twoheadrightarrow_{\widehat{\beta}} S\mathcal{C}_k[\![E\langle v\rangle]\!]$.

Proof: The proof is by case analysis of E. We elide S in the steps below as it remains constant.

- Case $E\langle\textsf{\textbf{ref}}\ \bullet\rangle$.

$$\begin{array}{lll} & \mathcal{K}_k[\![E\langle\textsf{\textbf{ref}}\ \bullet\rangle]\!]\texttt{(}\Phi[\![v]\!]\texttt{)} & \\ = & \texttt{(}\textsf{\textbf{func}}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle\textsf{\textbf{ref}}\ x\rangle]\!]\ \texttt{\})(}\Phi[\![v]\!]\texttt{)} & \\ \rightarrow & \mathcal{C}_k[\![E\langle\textsf{\textbf{ref}}\ x\rangle]\!][x/\Phi[\![v]\!]] & \widehat{\beta} \\ = & \mathcal{C}_k[\![E\langle\textsf{\textbf{ref}}\ x\rangle[x/v]]\!] & \text{by lemma 23} \\ = & \mathcal{C}_k[\![E\langle\textsf{\textbf{ref}}\ v\rangle]\!] & \end{array}$$

- Case $E\langle\textsf{\textbf{break}}\ \mathit{label}\ \bullet\rangle$.

$$\begin{array}{lll} & \mathcal{K}_k[\![E\langle\textsf{\textbf{break}}\ \mathit{label}\ \bullet\rangle]\!]\texttt{(}\Phi[\![v]\!]\texttt{)} & \\ = & \texttt{(}\textsf{\textbf{func}}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle\textsf{\textbf{break}}\ \mathit{label}\ x\rangle]\!]\ \texttt{\})(}\Phi[\![v]\!]\texttt{)} & \\ \rightarrow & \mathcal{C}_k[\![E\langle\textsf{\textbf{break}}\ \mathit{label}\ x\rangle]\!][x/\Phi[\![v]\!]] & \widehat{\beta} \\ = & \mathcal{C}_k[\![E\langle\textsf{\textbf{break}}\ \mathit{label}\ x\rangle[x/v]]\!] & \text{by lemma 23} \\ = & \mathcal{C}_k[\![E\langle\textsf{\textbf{break}}\ \mathit{label}\ v\rangle]\!] & \end{array}$$

- Case $E\langle\textsf{\textbf{if}}\ \texttt{(}\bullet\texttt{)}\ \texttt{\{}\ M_1\ \texttt{\}}\ \textsf{\textbf{else}}\ \texttt{\{}\ M_2\ \texttt{\}}\rangle$.

$$\begin{array}{lll} & \mathcal{K}_k[\![E\langle\textsf{\textbf{if}}\ \texttt{(}\bullet\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textsf{\textbf{else}}\ \texttt{\{}\ e_3\ \texttt{\}}\rangle]\!]\texttt{(}\Phi[\![v]\!]\texttt{)} & \\ = & \textsf{\textbf{func}}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle\textsf{\textbf{if}}\ \texttt{(}x\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textsf{\textbf{else}}\ \texttt{\{}\ e_3\ \texttt{\}}\rangle]\!]\ \texttt{\}(}\Phi[\![v]\!]\texttt{)} & \\ \rightarrow & \mathcal{C}_k[\![E\langle\textsf{\textbf{if}}\ \texttt{(}x\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textsf{\textbf{else}}\ \texttt{\{}\ e_3\ \texttt{\}}\rangle]\!][x/\Phi[\![v]\!]] & \widehat{\beta} \\ = & \mathcal{C}_k[\![E\langle\textsf{\textbf{if}}\ \texttt{(}x\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textsf{\textbf{else}}\ \texttt{\{}\ e_3\ \texttt{\}}\rangle[x/v]]\!] & \text{lemma 23} \\ = & \mathcal{C}_k[\![E\langle\textsf{\textbf{if}}\ \texttt{(}v\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textsf{\textbf{else}}\ \texttt{\{}\ e_3\ \texttt{\}}\rangle]\!] & \end{array}$$

- Case $E\langle \bullet\texttt{(}e\cdots\texttt{)}\rangle$.

$$\begin{array}{lll} & \mathcal{K}_k[\![E\langle \bullet\texttt{(}e\cdots\texttt{)}\rangle]\!]\texttt{(}\Phi[\![v]\!]\texttt{)} & \\ = & \textbf{func}\texttt{(}f\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle f\texttt{(}e\cdots\texttt{)}\rangle]\!]\ \texttt{\}(}\Phi[\![v]\!]\texttt{)} & \\ \rightarrow & \mathcal{C}_k[\![E\langle f\texttt{(}e\cdots\texttt{)}\rangle]\!][f/\Phi[\![v]\!]] & \widehat{\beta_v} \\ = & \mathcal{C}_k[\![E\langle v\texttt{(}e\cdots\texttt{)}\rangle]\!] & \text{lemma 23} \end{array}$$

Cases $E\langle v_f\texttt{(}v\cdots\bullet\, e\cdots\texttt{)}\rangle$ and $E\langle op_n\texttt{(}v\cdots\bullet\, e\cdots\texttt{)}\rangle$ are similar.

- Case $E\langle label\texttt{:}\ \texttt{\{}\ \bullet\ \texttt{\}}\rangle$.

$$\begin{array}{lll} & \mathcal{K}_k[\![E\langle label\texttt{:\{}\ \bullet\ \texttt{\}}\rangle]\!]\texttt{(}\Phi[\![v]\!]\texttt{)} & \\ = & \textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle label\texttt{:}\ \texttt{\{}\ x\ \texttt{\}}\ \rangle]\!]\ \texttt{\}(}\Phi[\![v]\!]\texttt{)} & \\ \rightarrow & \mathcal{C}_k[\![E\langle label\texttt{:}\ \texttt{\{}\ x\ \texttt{\}}\ \rangle]\!]\texttt{(}\Phi[\![v]\!]\texttt{)} & \widehat{\beta_v} \\ = & \mathcal{C}_k[\![E\langle label\texttt{:}\ \texttt{\{}\ v\ \texttt{\}}\ \rangle]\!] & \text{lemma 23} \end{array}$$

- Case $E\langle \textbf{deref}\ \bullet\rangle$.

$$\begin{array}{lll} & \mathcal{K}_k[\![E\langle \textbf{deref}\ \bullet\rangle]\!]\texttt{(}\Phi[\![v]\!]\texttt{)} & \\ = & \texttt{(}\textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle \textbf{deref}\ x\rangle]\!]\ \texttt{\})(}\Phi[\![v]\!]\texttt{)} & \\ \rightarrow & \mathcal{C}_k[\![E\langle \textbf{deref}\ x\rangle]\!][x/\Phi[\![v]\!]] & \widehat{\beta_v} \\ = & \mathcal{C}_k[\![E\langle \textbf{deref}\ x\rangle[x/v]]\!] & \text{by lemma 23} \\ = & \mathcal{C}_k[\![E\langle \textbf{deref}\ v\rangle]\!] & \end{array}$$

- Case $E\langle \textsf{setref}\ \bullet\ e\rangle$.

$$\begin{array}{lll} & \mathcal{K}_k[\![E\langle \textsf{setref}\ \bullet\ e\rangle]\!]\texttt{(}\Phi[\![v]\!]\texttt{)} & \\ = & \texttt{(}\textbf{func}\texttt{(}x\texttt{)}\ \texttt{\{}\ \mathcal{C}_k[\![E\langle \textsf{setref}\ x\ e\rangle]\!]\ \texttt{\})(}\Phi[\![v]\!]\texttt{)} & \\ \rightarrow & \mathcal{C}_k[\![E\langle \textsf{setref}\ x\ e\rangle]\!][x/\Phi[\![v]\!]] & \widehat{\beta_v} \\ = & \mathcal{C}_k[\![E\langle \textsf{setref}\ x\ e\rangle[x/v]]\!] & \text{by lemma 23} \\ = & \mathcal{C}_k[\![E\langle \textsf{setref}\ v\ e\rangle]\!] & \end{array}$$

Case $E\langle \textsf{setref}\ v\ \bullet\rangle$ is similar.

- Case $E\langle\textsf{tagcheck}\ R\ \bullet\rangle$.

$$\begin{array}{lll}
 & \mathcal{K}_k[\![E\langle\textsf{tagcheck}\ R\ \bullet\rangle]\!]\texttt{(}\Phi[\![v]\!]\texttt{)} & \\
= & \texttt{(}\textbf{func}\texttt{(}x\texttt{) \{ }\mathcal{C}_k[\![E\langle\textsf{tagcheck}\ R\ x\rangle]\!]\texttt{ \})(}\Phi[\![v]\!]\texttt{)} & \\
\to & \mathcal{C}_k[\![E\langle\textsf{tagcheck}\ R\ x\rangle]\!][x/\Phi[\![v]\!]] & \widehat{\beta_v} \\
= & \mathcal{C}_k[\![E\langle\textsf{tagcheck}\ R\ x\rangle[x/v]]\!] & \text{by lemma 23} \\
= & \mathcal{C}_k[\![E\langle\textsf{tagcheck}\ R\ v\rangle]\!] &
\end{array}$$

■

Lemma 23 (Substitution Commutes with $\mathcal{C}_k$) $\mathcal{C}_k[\![e]\!][x/\Phi(v)] = \mathcal{C}_k[\![e[x/v]]\!]$.

Proof: by induction on the structure of e. For each inductive case of $\mathcal{C}_k$, expand the definitions of Φ and $\mathcal{K}_k$ on the RHS to get a term with $\mathcal{C}_k$ applied to subterms of e. The inductive hypotheses now trivially apply. ■

C.4 Additional Rules for Flow Analysis

For soundness of flow analysis, we need to reason about intermediate terms with concrete locations, l. Hence, we extend abstract heaps to the type:

$\boxed{\widehat{S} : \hat{l} + l \to R}$

$\boxed{\widehat{S}; \widehat{\Gamma} \triangleright V \rightsquigarrow \widehat{V}}$

$$\frac{\widehat{S}(\hat{l}) = R \qquad \mathit{runtime}(V) = R}{\widehat{S}; \widehat{\Gamma} \triangleright V \rightsquigarrow \textsf{Deref}\ \hat{l}\ R} \quad (\text{V-Deref})$$

$$\frac{\widehat{S}; \widehat{\Gamma} \triangleright V \rightsquigarrow \textsf{Deref}\ \hat{l}}{\widehat{S}; \widehat{\Gamma} \triangleright \delta_1(\textbf{typeof}, V) \rightsquigarrow \textsf{LocTypeof}\ \hat{l}} \quad (\text{V-Typeof})$$

$$\frac{\widehat{S}; \widehat{\Gamma} \triangleright V_1 \rightsquigarrow \textsf{LocType}\ \hat{l} \qquad V_2 \in \widehat{S}(\hat{l})}{\widehat{S}; \widehat{\Gamma} \triangleright \delta_2(\texttt{===}, V_1, V_2) \rightsquigarrow \textsf{LocType}\ \hat{l}\ R} \quad (\text{V-TypeIs})$$

$\boxed{\widehat{S}; \widehat{\Gamma} \vDash SM}$

$$\frac{\widehat{S};\widehat{\Gamma} \triangleright V \rightsquigarrow \widehat{V} \qquad \widehat{S}; x : r, \widehat{\Gamma} \vDash S\ M}{\widehat{S};\widehat{\Gamma} \vDash S\ \textsf{let}\ (x\ \texttt{=}\ \textsf{deref}\ V)\ M} \quad \text{(F-Deref-Unk)}$$

$$\frac{\widehat{S};\widehat{\Gamma} \triangleright V \rightsquigarrow \widehat{V} \qquad \widehat{S}; x : \{\text{"string"}\}, \widehat{\Gamma} \vDash S\ M}{\widehat{S};\widehat{\Gamma} \vDash S\ \textsf{let}\ (x\ \texttt{=}\ \textsf{typeof}\ V)\ M} \quad \text{(F-Typeof-Unk)}$$

$$\frac{\widehat{S};\widehat{\Gamma} \triangleright V_1 \rightsquigarrow \widehat{V_1} \qquad \widehat{S};\widehat{\Gamma} \triangleright V_2 \rightsquigarrow \widehat{V_2} \qquad \widehat{S}; x : \{\text{"boolean"}\}, \widehat{\Gamma} \vDash S\ M}{\widehat{S};\widehat{\Gamma} \vDash S\ \textsf{let}\ (x\ \texttt{=}\ V_1\ \texttt{===}\ V_2)\ M} \quad \text{(F-Eq)}$$

$$\frac{\widehat{S};\widehat{\Gamma} \triangleright V_1 \rightsquigarrow \mathsf{Ref}\ \hat{l}_1 \qquad \widehat{S};\widehat{\Gamma} \triangleright V_2 \rightsquigarrow \mathsf{Ref}\ \hat{l}_2 \qquad \hat{l} : R, \widehat{S}; x : r, del(\hat{l}_1, del(\hat{l}_2, \widehat{\Gamma})) \vDash S\ M}{\widehat{S};\widehat{\Gamma} \vDash S\ \textsf{let}\ (x\ \texttt{=}\ \mathsf{setref}\ V_1\ V_2)\ M} \quad \text{(F-SetRef-Alias1)}$$

$$\frac{\widehat{S};\widehat{\Gamma} \triangleright V_1 \rightsquigarrow \widehat{V_1} \qquad \widehat{S};\widehat{\Gamma} \triangleright V_2 \rightsquigarrow \mathsf{Ref}\ \hat{l}_2 \qquad \hat{l} : R, \widehat{S}; x : r, del(\hat{l}_2, \widehat{\Gamma}) \vDash S\ M}{\widehat{S};\widehat{\Gamma} \vDash S\ \textsf{let}\ (x\ \texttt{=}\ \mathsf{setref}\ V_1\ V_2)\ M} \quad \text{(F-SetRef-Alias2)}$$

$$\frac{\widehat{S};\widehat{\Gamma} \triangleright V \rightsquigarrow \widehat{V} \qquad \widehat{S};\widehat{\Gamma} \vDash M_1 \qquad \widehat{S};\widehat{\Gamma} \vDash M_2}{\widehat{S};\widehat{\Gamma} \vDash S\ \textsf{if}\ (V)\ \{\ M_1\ \}\ \textsf{else}\ \{\ M_2\ \}} \quad \text{(F-If)}$$

$$\frac{\widehat{S};\widehat{\Gamma} \vDash M_1}{\widehat{S};\widehat{\Gamma} \vDash S\ \textsf{if}\ (\textsf{true})\ \{\ M_1\ \}\ \textsf{else}\ \{\ M_2\ \}} \quad \text{(F-If-True)}$$

$$\frac{\widehat{S};\widehat{\Gamma} \vDash M_2}{\widehat{S};\widehat{\Gamma} \vDash S\ \textsf{if}\ (\textsf{false})\ \{\ M_1\ \}\ \textsf{else}\ \{\ M_2\ \}} \quad \text{(F-If-False)}$$

C.5 Flow Analysis

Furthermore, flow analysis is sound:

Lemma 24 (Soundness of Flow Analysis) *If $\hat{S}, \cdot \vDash SM$ and $SM \rightarrow S'M'$, then either:*

- $\hat{S}', \cdot \vDash S'M'$, *or*

- *M is a $\widehat{\beta_v}$ redex,* $\textsf{func}(x \cdots)\ :\ T \cdots \bot\ \{\ N\ \}(V \cdots)$, *and for some V,* $\delta_1(\textsf{typeof}, V) \notin \mathit{runtime}(T)$.

Proof: by case-analysis on M.

- Case $S\ \textsf{func}(x \cdots)\ :\ T \cdots \bot\ \{\ N\ \}(V \cdots) \rightarrow SN[x/V] \cdots$.

  By inversion, $\cdot; x : \mathit{runtime}(T) \cdots \vDash N$ (V-Restart) and there exist $R \cdots$, such that $\cdot \triangleright V \rightsquigarrow R \cdots$. Since $V \cdots$ are all closed, $\delta_1(\textsf{typeof}, V) \cdots$ is defined. By case analysis on V, $\cdot; \cdot \triangleright V \rightsquigarrow \{\delta_1(\textsf{typeof}, V)\} \cdots$. There are two cases:

  - If $\{\delta_1(\textsf{typeof}, V)\} \sqsubseteq \mathit{runtime}(T) \cdots$, then by substitution (lemma 25), $\cdot; \cdot \vDash N[x/V \cdots]$.
  - Otherwise, for some V, T, $\delta_1(\textsf{typeof}, V) \notin \mathit{runtime}(T)$.

- Case $S\ \textsf{let}\ x\ =\ V \textsf{in}\ M \rightarrow SM[x/V]$.

  By F-LetVal, $\widehat{S}; \cdot \vDash S\ \textsf{let}\ x\ =\ V \textsf{in}\ M$. By hypothesis, $\widehat{S}; x : \widehat{V} \vDash SM$ and $\widehat{S}; \cdot \triangleright V \rightsquigarrow \widehat{V}$. By substitution (lemma 25), $\widehat{S}; \cdot \vDash SM[x/V]$.

- Case $S\ \textsf{let}\ x\ =\ \textsf{ref}\ V \textsf{in}\ M \rightarrow (l, V)S\ \textsf{let}\ x\ =\ l \textsf{in}\ M$.

  There are two applicable rules.

  - By F-Alloc, $\widehat{S}; \cdot \triangleright V \rightsquigarrow R$ and $\hat{l} : R, \widehat{S}; x : \textsf{Ref}\ \hat{l} \vDash SM$. By V-Loc, $(l, V), \widehat{S}; \cdot \triangleright l \rightsquigarrow \textsf{Ref}\ l$. Renaming $\hat{l}$ to l (lemma 26), $l : R, \widehat{S}; x : \textsf{Ref}\ l \vDash SM$. Hence by F-LetVal, $l : R; \widehat{S}; \cdot \vDash (l, V), S\ \textsf{let}\ x\ =\ l \textsf{in}\ M$.
  - By F-Ref-Alias, $\widehat{S}; \cdot \triangleright V \rightsquigarrow \textsf{Ref}\ \ \hat{l}$ and $\widehat{S}; x : r \vDash SM$. By V-Loc, followed by V-Sub, $(l, v), \widehat{S}; \cdot \triangleright \hat{l} \rightsquigarrow r$. Hence by F-LetVal, $l : r, \widehat{S}; \cdot \vDash (l, V), S\ \textsf{let}\ x\ =\ l \textsf{in}\ M$.

- Case $S\ \textsf{let}\ x\ =\ \textsf{deref}\ l\ \textsf{in}\ M \rightarrow S\ \textsf{let}\ x\ =\ S(l)\ \textsf{in}\ M$.

  - By F-Deref, $\widehat{S}; \cdot \triangleright l \rightsquigarrow \textsf{Ref}\ l$, $\widehat{S}(l) = R$, and $\widehat{S}; x : \textsf{Deref}\ l\ R \vDash S\ M$. By V-Deref, $\widehat{S}; \cdot \triangleright S(l) \rightsquigarrow \textsf{Deref}\ l\ R$. Apply F-LetVal to the RHS.
  - By F-Deref-Unk, $\widehat{S}; x : r \vDash SM$ and $\widehat{S}; \cdot \triangleright S(l) \rightsquigarrow \widehat{V}$. By V-Sub and F-LetVal, the conclusion follows.

- Case $S\ \textsf{let}\ x\ =\ \textsf{typeof}\ V\ \textsf{in}\ M \rightarrow S\ \textsf{let}\ x\ =\ \delta_1(\textsf{typeof}, V)\ \textsf{in}\ M$

  - By F-Typeof, $\cdot \triangleright V \rightsquigarrow \textsf{Deref}\ \hat{l}\ R$ and $\widehat{S}; x : \textsf{LocTypeof}\ \hat{l} \vDash M$. By V-Typeof, $\widehat{S}; \cdot \triangleright \delta_1(\textsf{typeof}, V) \rightsquigarrow \textsf{LocTypeof}\ \hat{l}$. Apply F-LetVal to the RHS.

  - By F-Typeof-Unk, $\widehat{S}; x : \{\text{"string"}\} \vDash SM$ and $\widehat{S}; \cdot \rhd V \rightsquigarrow \widehat{V}$. By definition of $\delta_1(\textbf{typeof})$, $\widehat{S}; \cdot \rhd \delta_1(\textbf{typeof}, V) \rightsquigarrow \{\text{"string"}\}$. Apply F-LetVal.

- Case S **let** x = (V_1 === V_2) in $M \to S$ S **let** x = δ_2(===, V_1, V_2) in M.

  - By F-TypeIs-Str, $V_2 = \text{"string"}$, $\widehat{S}; \cdot \rhd V_1 \rightsquigarrow \mathsf{LocTypeof}\ \hat{l}$, and $\widehat{S}; x : \mathsf{LocType}\ \hat{l}\ \{\text{"string"}\} \vDash M$. By V-TypeIs, $\widehat{S}; \cdot \rhd \delta_2(\text{===}, V_1, V_2) \rightsquigarrow \mathsf{LocTypeof}\ \hat{l}$. Apply F-LetVal to the RHS.
  - By F-Eq, $\widehat{S}; x : \{\text{"boolean"}\} \vDash SM$, $\widehat{S}; \cdot \rhd V_1 \rightsquigarrow \widehat{V_1}$, and $\widehat{S}; \cdot \rhd V_2 \rightsquigarrow \widehat{V_2}$. By definition of $\delta_2(\text{===})$, $\widehat{S}; \cdot \rhd \delta_2(\text{===}, V_1, V_2) \rightsquigarrow \{\text{"boolean"}\}$. Apply F-LetVal.

- Case S **let** x = tagcheck R V in $M \to S$ **let** x = V in M.

  By F-TagCheck, $\widehat{S}; \cdot \rhd V \rightsquigarrow R$. Apply F-LetVal to the RHS.

- Case S **let** x = setref l V in M.

  - By F-SetRef, $\widehat{S}; \cdot \rhd l \rightsquigarrow \mathsf{Ref}\ l$, $\widehat{S}; \cdot \rhd V \rightsquigarrow R$, and $\widehat{S}[l := R], x : \mathsf{Ref}\ l \vDash SM$. Apply F-LetVal.
  - By F-SetRef-Alias2, $\widehat{S}; \cdot \rhd l \rightsquigarrow \mathsf{Ref}\ l$, $\widehat{S}; \cdot \rhd V \rightsquigarrow \mathsf{Ref}\ \hat{l}'$, and $\widehat{S}; x : r \vDash SM$. Apply F-LetVal.

- Case S **if** (**true**) { M_1 } **else** { M_2 } $\to S$ M_1.

  By F-If-True, $\widehat{S}; \cdot \rhd V \rightsquigarrow \mathsf{LocType}\ \hat{l}\ R$, $R \sqsubseteq \widehat{S}(\hat{l})$, and $\widehat{S}; \cdot \vDash M_1$.

- Case S **if** (**false**) { M_1 } **else** { M_2 } $\to S$ M_2.

  By F-If-False, $\widehat{S}; \cdot \rhd V \rightsquigarrow \mathsf{LocType}\ \hat{l}\ R$, $R \sqsubseteq \widehat{S}(\hat{l})$, and $\widehat{S}; \cdot \vDash M_2$.

Lemma 25 (Substitution) *If $\widehat{S}; x : \widehat{V}, \widehat{\Gamma} \vDash M$ and $\widehat{S}; \widehat{\Gamma} \rhd V \rightsquigarrow \widehat{V'}$, with $\widehat{V'} \sqsubseteq \widehat{V}$ then $\widehat{S}; \widehat{\Gamma} \vDash M[x/V']$.*

Proof by induction on $\widehat{S}; x : \widehat{V}, \widehat{\Gamma} \vDash M$.

The key to this proof is substituting values W that occur in expressions M. i.e., we must show that if $\widehat{S}; x : \widehat{V}, \widehat{\Gamma} \rhd W \rightsquigarrow \widehat{W}$, then $\widehat{S}; \widehat{\Gamma} \rhd W[x/V'] \rightsquigarrow \widehat{W}$.

The interesting case is when $W = \textbf{func}(y \cdots) : T \cdots \to \bot$ { M }. So, by V-Restart:

$$\begin{aligned} & \cdot; y : runtime(T) \cdots, reset(x : \widehat{V}, \widehat{\Gamma}) \vDash M' \\ = \quad & \cdot; reset(x : \widehat{V''}), y : runtime(T) \cdots, reset(\widehat{\Gamma}) \vDash M' \end{aligned}$$

By definition of $reset$, $\widehat{V} \sqsubseteq \widehat{V''}$. Since M' is smaller than M, the inductive hypothesis applies.

Lemma 26 (Renaming Locations) *If* $\widehat{l} : \widehat{V}, \widehat{S}; \widehat{\Gamma} \vDash SM$ *and* $l \notin dom(S)$*, then* $l : \widehat{V}, \widehat{S}; \widehat{\Gamma}[\widehat{l}/l] \vDash SM$*.*

Proof by induction on the size of $\widehat{\Gamma}$. ■

C.6 Combined Soundness Theorems

Theorem 8 (Strengthened Progress) *If:*

i. $\Sigma; \cdot \vdash e : T$,

ii. $\Sigma \vdash \sigma$, *and*

iii. $\widehat{S}; \cdot \vDash \mathcal{P}_k[\![\sigma e]\!]$,

then either:

i. $e \in v$, *or*

ii. There exist σ' *and* e'*, such that* $\sigma e \rightarrow \sigma' e'$*.*

Proof: This follows from lemma 6, with the possibility of tagerrs eliminated by inspection of the acceptability relation—flow analysis does not admit expressions with tagerrs. ■

Theorem 9 (Combined Preservation) *If:*

i. $\Sigma; \cdot \vdash e : T$,

ii. $\Sigma \vdash \sigma$,

ii. $\widehat{S}; \cdot \vDash \mathcal{P}_k[\![\sigma e]\!]$, *and*

iv. $\sigma e \rightarrow \sigma' e'$,

then there exist Σ' *and* $\widehat{S}'$*, such that:*

i. $\Sigma'; \cdot \vdash e' : T$,

ii. $\Sigma' \vdash \sigma'$,

iii. $\Sigma \subseteq \Sigma'$, *and* *iv.* $\hat{S}'; \cdot \vDash \mathcal{P}_k[\![\sigma' e']\!]$.

Proof: Conclusions (i.), (ii.), and (iii.) follow immediately from lemma 5. For conclusion (iv.), apply lemma 7 to hypothesis (iv.) to get a reduction sequence, $\mathcal{P}_k[\![\sigma e]\!] \twoheadrightarrow \mathcal{P}_k[\![\sigma' e']\!]$. Apply lemma 8 at each step, eliminating case (ii.) of the lemma as follows. By lemma 7, intermediate expressions are not β_v-redexes, so case (ii.) does not apply. Suppose e itself has an active β_v-redex:

$$e = E\langle \textsf{func}\texttt{(}x \cdots\texttt{)} \ \texttt{:} \ U \cdots \rightarrow S \ \texttt{\{} \ e_f \ \texttt{\}(}v \cdots\texttt{)}\rangle$$

Transform e to CPS:

$$\mathcal{C}_k[\![e]\!] \quad = \quad \Phi[\![\textsf{func}\texttt{(}x \cdots\texttt{)} \ \texttt{:} \ U \cdots \rightarrow S \ \texttt{\{} \ e_f \ \texttt{\}}]\!]\texttt{(}\mathcal{K}_k[\![E]\!]\texttt{,}\Phi[\![v]\!] \cdots\texttt{)}$$

Since e is typable, there exists a Γ such that:

$$\Sigma; \Gamma \vdash \textsf{func}\texttt{(}x \cdots\texttt{)} \ \texttt{:} \ U \cdots \rightarrow S \ \texttt{\{} \ e_f \ \texttt{\}(}v \cdots\texttt{)} : S$$

Furthermore, by inversion (lemma 17), $\Sigma; \Gamma \vdash v : U \cdots$. For all v, $\delta_1(\textsf{typeof}, v) \in \mathit{runtime}(U)$ (lemma 20). By inspection of Φ, $\delta_1(\textsf{typeof}, v) = \delta_1(\textsf{typeof}, \Phi[\![v]\!])$, case (ii.) of lemma 8 does not apply. ■

Appendix D

Proofs: Fluid Object Types

D.1 Definitions

Definition 3 (Type Equivalence) *We define a relation on types $=_T$.*

$$\frac{M_A \subseteq L_A \qquad L_A \subseteq M_A \qquad \forall i, j. L_i \cap M_j \neq \emptyset \Rightarrow S_i =_T T_j \wedge p_i = q_j \qquad \forall i. \exists (j_1, \cdots, j_k). L_i \subseteq \bigcup_{l \in (j_1, \cdots, j_k)} M_l \qquad \forall j. \exists (i_1, \cdots, i_k). M_i \subseteq \bigcup_{l \in (i_1, \cdots, i_k)} L_l}{\{L_1^{p_1} : S_1, \cdots, L_n^{p_n} : S_n, L_A : \mathbf{abs}\} =_T \{M_1^{q_1} : T_1, \cdots, M_m^{q_m} : T_m, M_A : \mathbf{abs}\}} \text{(Equiv-Obj)}$$

The other cases of $=_T$ are trivial; to define them we lift Equiv-Obj in the natural way over the other types. $=_T$ describes an equivalence class of types—we use $T_1 =_T T_2$ and $T_1 = T_2$ interchangeably in this document, and types represented by the same letter are assumed to be related by $=_T$.

Definition 4 (Subtyping) *The generating function,*

$$\mathcal{ST} : \mathcal{P}(\Gamma \times \mathcal{T} \times \mathcal{T} + p \times p) \to \mathcal{P}(\Gamma \times \mathcal{T} \times \mathcal{T} + p \times p)$$

is defined co-inductively by the subtyping judgments. We define subtyping as $\Gamma \vdash S <: T$, iff $(\Gamma, S, T) \in \nu\mathcal{ST}$ and $p <: q$, iff $(p, q) \in \nu\mathcal{ST}$.

Definition 5 (Transitivity) *For $R \subseteq \mathcal{P}(\Gamma \times \mathcal{T} \times \mathcal{T} + p \times p)$*

$$\begin{array}{rcl} TR(R) & = & \{(\Gamma, x, z) \mid \forall x, z \in \mathcal{T}, \exists y \in \mathcal{T}, (\Gamma, x, y), (\Gamma, y, z) \in R\} \\ & \cup & \{(x, z) \mid \forall x, z \in p, \exists y \in p, (x, y), (y, z) \in R\} \end{array}$$

Definition 6 (Top-down Subexpressions) *S is a top-down subexpression of T, written $S \sqsubseteq T$, if (S, T) is in μTD, defined as follows:*

$$\begin{array}{rcl} TD(R) & = & \{(T,T) \mid T \textit{ is a finite type } \} \\ & \cup & \{(S, \{L^p : T, rest \cdots\}) \mid (S, T) \in R\} \\ & \cup & \{(S, \{L^p : T, rest \cdots\} \mid (S, \{rest\}) \in R\} \\ & \cup & \{(S, \{L^p : T\}) \mid (S, T) \in R\} \\ & \cup & \{(S, \textsf{Ref}\, T) \mid (S, T) \in R\} \\ & \cup & \{(S, \mu x.T) \mid (S, T[x/\mu x.T]) \in R\} \\ & \cup & \{(S, T_1 \to T_2) \mid (S, T_1) \in R\} \\ & \cup & \{(S, T_1 \to T_2) \mid (S, T_2) \in R\} \\ & \cup & \{(S, \forall \alpha <: U.T) \mid (S, U) \in R\} \\ & \cup & \{(S, \forall \alpha <: U.T) \mid (S, T) \in R\} \end{array}$$

Definition 7 (Bottom-Up Subexpressions) *S is a bottom-up subexpression of T, written $S \preceq T$, if (S, T) is in μBU, defined as follows:*

$$\begin{array}{rcl} BU(R) & = & \{(T,T) \mid T \textit{ is a finite type } \} \\ & \cup & \{(S, \{L^p : T, rest \cdots\}) \mid (S, T) \in R\} \\ & \cup & \{(S, \{L^p : T, rest \cdots\} \mid (S, \{rest\}) \in R\} \\ & \cup & \{(S, \{L^p : T\}) \mid (S, T) \in R\} \\ & \cup & \{(S, \textsf{Ref}\, T) \mid (S, T) \in R\} \\ & \cup & \{(S, T_1 \to T_2) \mid (S, T_1) \in R\} \\ & \cup & \{(S, T_1 \to T_2) \mid (S, T_2) \in R\} \\ & \cup & \{(S, \forall \alpha <: U.T) \mid (S, U) \in R\} \\ & \cup & \{(S, \forall \alpha <: U.T) \mid (S, T) \in R\} \\ & \cup & \{(S[x/\mu x.T], \mu x.T) \mid (S, T) \in R\} \end{array}$$

Definition 8 (Active Expressions) *Active expressions, ae, are defined as follows:*

$$
\begin{array}{rcl}
ae & = & v_1\texttt{(}v_2\texttt{)} \\
 & | & \textbf{ref}\ v \\
 & | & \textbf{deref}\ v \\
 & | & v_1\ \texttt{=}\ v_2 \\
 & | & v_1\texttt{[}v_2\texttt{]} \\
 & | & v_1\texttt{[}v_2\ \texttt{=}\ v_3\texttt{]} \\
 & | & \textbf{delete}\ v_1\texttt{[}v_2\texttt{]} \\
 & | & \textbf{if}\ \texttt{(}v_1\texttt{)}\ \texttt{\{}\ e_2\ \texttt{\}}\ \textbf{else}\ \texttt{\{}\ e_3\ \texttt{\}} \\
 & | & v_1\ \textbf{hasfield}\ v_2 \\
 & | & v_1\ \textbf{matches}\ v_2 \\
 & | & \textbf{fieldin}\ \texttt{\{}\ s_1 : v_1\texttt{,}\ s_2 : v_2\ \cdots\texttt{\}}\ \textsf{init}\ v_{acc}\ \textbf{do}\ v_f
\end{array}
$$

D.2 Auxilliary Lemmas

Lemma 27 *For all T, $\mathcal{T}_T^{\downarrow} \subseteq \mathcal{T}_T^{\uparrow}$*

Proof This is exactly the same argument as Pierce [71]. ■

Lemma 28 *The set of top-down subexpressions, $\mathcal{T}_T^{\downarrow} = \{S \mid (S,T) \in \mu TD\}$, is finite for all T.*

Proof By lemma 27, $\mathcal{T}_T^{\downarrow} \subseteq \mathcal{T}_T^{\uparrow}$ for all T, and by lemma 29, $\mathcal{T}_T^{\uparrow}$ is finite for all T, so $\mathcal{T}_T^{\downarrow}$ is finite for all T.

Lemma 29 *The set of bottom-up subexpressions, $\mathcal{T}_T^{\uparrow} = \{S \mid (S,T) \in \mu BU\}$ is finite for all T.*

Proof The second position in the each right-hand clause in BU is smaller than the left.

Lemma 30 *If $(S, T[x/U]) \in \mathcal{T}_{T[x/U]}^{\uparrow}$, then either $(S,U) \in \mathcal{T}_U^{\uparrow}$ or $S = S'[x/U]$ for some $(S',T) \in \mathcal{T}_T^{\uparrow}$.*

Proof By case analysis on T.

D.3 Subtyping

Lemma 31 *If:*

$$\mathcal{S} : \mathcal{P}(\Gamma \times \mathcal{T} \times \mathcal{T} + p \times p) \to \mathcal{P}(\Gamma \times \mathcal{T} \times \mathcal{T} + p \times p)$$

is a monotone function, and for all R, $TR(\mathcal{S}(R)) \subseteq \mathcal{S}(TR(R))$, then $\nu\mathcal{S}$ is transitive.

Proof: This definition is from Gapayev, et al., and reproduced in Pierce's text. Its relation to transitivity is discussed there—we use it as a goal and defer to their explanation to complete the proof [71, Lemma 21.3.6].

Lemma 32 (Subtyping is Transitive) $TR(\nu\mathcal{ST}) \subseteq \nu\mathcal{ST}$

Proof:

For arbitrary R, we consider both field annotations in (p, q), and subtyping judgments (Γ, S, T):

Let $(p, q) \in TR(\mathcal{ST}(R))$. By definition of TR, there exists a p' such that $(p, p'), (p', q) \in \mathcal{ST}(R)$. By case analysis of the $p <: p$ rules, it follows trivially that $(p, q) \in \mathcal{ST}(TR(R))$.

Let $(\Gamma, S, T) \in TR(\mathcal{ST}(R))$. By definition of TR, there exists a U such that $(\Gamma, S, U), (\Gamma, U, T) \in \mathcal{ST}(R)$. We will show that $(\Gamma, S, T) \in \mathcal{ST}(TR(R))$, so that by lemma 31, $\nu\mathcal{ST}$ is transitive.

By case-analysis on the possible shapes of U (eliding the trivial cases where $T = \top$):

- $U = \{L_1^{p_1} : U_1, \cdots, L_n^{p_n} : U_n, L_A : \mathbf{abs}\}$.

  Since $(\Gamma, S, U), (\Gamma, U, T) \in \mathcal{ST}(R)$, by cases of $\mathcal{ST}$,

  (1) $S = \{K_1^{o_1} : S_1, \cdots, K_l^{o_l} : S_l, K_A : \mathbf{abs}\}$, and

  (2) $T = \{M_1^{q_1} : T_1, \cdots, M_m^{q_m} : T_m, M_A : \mathbf{abs}\}$.

  By hypothesis, $(\Gamma, S, U) \in \mathcal{ST}(R)$, which must be by S-Object. By hypothesis of S-Object:

  (3) $\forall i, j.$ if $K_i \cap L_j \neq \emptyset$ then $(S_i, U_j) \in R$ and $(o_i, p_j) \in R$,

  (4) $\bigcup_i^{1\cdots n} L_i \subseteq \bigcup_h^{1\cdots l} K_h \cup K_A$,

  (4') $L_A \subseteq K_A$,

  (5) $\forall i.$ if $L_i \cap K_A \neq \emptyset$ then $(p_j = \circ$ or $p_j = \uparrow)$,

  (6) $\forall i.$ if $p_i = \uparrow$ then $(\Gamma, \mathit{inherit}(S, L_i), U_i) \in R$.

  Similarly,

(7) $\forall i, j.$ if $L_i \cap M_j \neq \emptyset$ then $(U_i, T_j) \in R$ and $(p_i, q_j) \in R$,

(8) $\bigcup_j^{1\cdots m} M_j \subseteq \bigcup_i^{1\cdots n} L_i \cup L_A$,

(8') $M_A \subseteq L_A$,

(9) $\forall j.$ if $M_j \cap L_A \neq \emptyset$ then $(q_j = \circ$ or $q_j =\uparrow)$,

(10) $\forall j.$ if $q_j =\uparrow$ then $(\Gamma, \mathit{inherit}(U, M_j), T_j) \in R$.

Our goal is to show that $(\Gamma, S, T) \in \mathcal{ST}(TR(R))$, by constructing a proof of S-Object using the above hypotheses and the definition of $\mathcal{ST}$ and TR. Informally, we need to find support for the hypotheses of S-Object for S and T among the elements of $TR(R)$. In particular, we show that:

a. $\bigcup_j^{1\cdots m} M_j \subseteq \bigcup_h^{1\cdots l} K_h \cup K_A$.

Proof: By (4), (4'), (8), (8'), and transitivity of subset inclusion.

b. $\forall h, j.$ if $K_h \cap M_j \neq \emptyset$ then $(o_h, q_j) \in TR(R)$ and $(\Gamma, S, T) \in TR(R)$

Proof: Let $x \in K_h \cap M_j$, thus $x \in K_h$ and $x \in M_j$. By (8), there are two cases:

i. $x \in L_A$ — In this case, $M_j \cap L_A \neq \emptyset$. Since $L_A \subseteq K_A$ by (8'), $x \in K_A$. But by the well-formedness of object types, object types' fields are partitioned, and this is a contradiction, since $x \in K_h$. This case cannot occur.

ii. $x \in L_i$ for some i — In this case, we have that $x \in L_i$ and $x \in M_j$, so by (7), $(U_i, T_j) \in R$ and $(p_i, q_j) \in R$. We also have that $x \in L_i$ and $x \in K_h$, so by (3), $(S_h, U_i) \in R$ and $(o_h, p_i) \in R$. This completes item b., as by definition of TR, $(o_h, p_i) \in R, (p_i, q_j) \in R \Rightarrow (o_h, q_j) \in TR(R)$, and $(S_h, U_i) \in R, (U_i, T_j) \in R \Rightarrow (S_h, T_j) \in TR(R)$.

c. $\forall j. M_j \cap K_A \neq \emptyset \Rightarrow (q_j = \circ$ or $q_j =\uparrow)$

Proof: For each non-vacuous case of j, there is some x with $x \in M_j$ and $x \in K_A$. By (8), there are two cases:

i. $x \in M_j \cap L_i$ for some i — In this case, $L_i \cap K_A \neq \emptyset$, so by (5) $p_i = \circ$ or $p_i =\uparrow$. Only $p-$Refl applies, so item c. is complete.

ii. $x \in M_j \cap L_A$ — This case follows directly from (9).

d. $\forall j$. if $q_j = \uparrow$ then $(\Gamma, \mathit{inherit}(S, M_j), T_j) \in TR(R)$.

Proof: By (10), $\forall j$. if $q_j = \uparrow$ then $(\Gamma, \mathit{inherit}(U, M_j), T_j) \in R$. By assumption, we have that $(\Gamma, S, U) \in R$ (or, equivalently, $\Gamma \vdash S <: U$). Recall from (1) that $S = \{K_1^{o_1} : S_1, \cdots, K_l^{o_l} : S_n, K_A : \mathbf{abs}\}$. By lemma 38, since $\Gamma \vdash S <: U, M_j \subseteq M_j$, it must be that $\Gamma \vdash \mathit{inherit}(S, M_j) <: \mathit{inherit}(U, M_j)$ for each j. Now we have that $(\Gamma, \mathit{inherit}(S, M_j), \mathit{inherit}(U, M_j)) \in R$ and $(\Gamma, \mathit{inherit}(U, M_j), T_j) \in R$ for each j, which is sufficient to show that $(\Gamma, \mathit{inherit}(S, M_j), T_j) \in TR(R)$ for each j, which completes the proof.

- $U = b$ Only S-b applies, so S and T must both be b, and are therefore in R.

- **Case** $U = L_u$**.** Only case S-Str applies for $S <: U$ and $U <: T$. Thus, $S = L_s$ and $T = L_T$, with $L_s \subseteq L_u \subseteq L_T$. Thus $S <: T$ follows by transitivity of the subset relation.

- **Case** $U = \textbf{Ref}\ U'$**.** Only case S-Ref applies, thus $S = \mathsf{Ref}\ S'$ and $T = \mathsf{Ref}\ T'$, with

$$(S', U'), (U', S'), (U', T'), (T', U') \in R$$

By definition of TR,

$$(S', T'), (T', S') \in TR(R)$$

Thus,

$$(\mathsf{Ref}\ S', \mathsf{Ref}\ T') \in \mathcal{ST}(TR(R))$$

- **Case** $U = \alpha$**.** There are three possibilities, depending on uses of S-VR and S-VTR:
  - $S = \alpha$ and $T = \alpha$, so $(\Gamma, S, T) \in TR(R)$ by S-VR.
  - $S = \beta$, $\beta \neq \alpha$, $(\beta <: \alpha) \in \Gamma$, and $(\alpha <: T) \in \Gamma$. In this case, $(\Gamma, S, T) \in TR(R)$ by S-VTR.
  - $S = \alpha$ and $(\alpha <: T) \in \Gamma$. In this case, $(\Gamma, S, T) \in TR(R)$ by S-VTR.

- **Case** $U = \forall \alpha <: U_1.U_2$. The only rule that applies is S-Kern, so it must be that:
  - $S = \forall \alpha <: U_1.S_2$,
  - $((\Gamma, \alpha <: U_1), S_2, U_2) \in \mathcal{ST}(R)$,

- $T = \forall\alpha <: U_1.T_2$,
- $((\Gamma, \alpha <: U_1), U_2, T_2) \in \mathcal{ST}(R)$.

By the definition of TR, $((\Gamma, \alpha <: U_1), S, T) \in TR(R)$.

- $U = \mu\alpha.T$ This case is addressed in [71, chapter 21].
- $U = U_1 \to U_2$ See [71, page 288]

■

Lemma 33 (Subtyping is Reflexive) *For all $T \in \mathcal{T}$, $(T, T) \in \nu\mathcal{ST}$, and for all $F \in \mathcal{F}$, $(F, F) \in \nu\mathcal{ST}$.*

Proof: By case analysis on the subtyping rules.

Lemma 34 *$\mathcal{ST}$ is Invertible*

Proof: The corresponding support function is well-defined. By inspection of the subtyping rules, for a given pair of expressions, only one typing rule applies.

Lemma 35 *For all types S and T, $S <: T$ is decidable.*

Proof: Since S and T are finite μ-types, the set *reachables*$_{\mathcal{ST}}(S, T)$ is finite [71, Proposition 21.9.11]. Thus, the algorithm *gfp*$_{\mathcal{ST}}$ [71, Definition 21.5.5] terminates [71, Theorem 21.5.12]. ■

Lemma 36 *For all Γ, T, L, M, $inherit_\Gamma(T, L) \sqcup inherit_\Gamma(T, M) = inherit_\Gamma(T, L \sqcup M)$.*

Proof: By induction on the syntactic size of T and by definition of the join operator.

Note that in the definition of *inherit*, the condition in both cases requires that for some L_C, $L_Q \sqsubseteq L_C$. $L \sqsubseteq L_C$ and $M \sqsubseteq L_C$ hold if the left-hand side of the equality are defined. However, $L \sqcup M \subseteq L_C$ holds only because $L \sqcup M = L \cup M$.

■

Lemma 37 *If $L_Q \subseteq M_Q \subseteq \bigcup_j^{1\cdots m} M_j$ and $\forall i, j. L_i \cap M_j \neq \emptyset \Rightarrow \Gamma \vdash S'_i <: T'_j$ then*

$$\Gamma \vdash \bigsqcup_i^{1\cdots n} \{S'_i \mid \Gamma \vdash L_Q \cap L_i \neq \emptyset\} <: \bigsqcup_j^{1\cdots m} \{T'_i \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset\}$$

Proof: It is sufficient to show that for all S'_i on the left-hand side, there exists a T'_j such that $\Gamma \vdash S'_i <: T'_j$.

For any S'_i, since $L_i \cap L_Q \neq \emptyset$, $\exists str.str \in L_i \cap L_Q$. Since $L_Q \subseteq \bigcup M_j$, intersecting on the left-hand side we have $L_i \cap L_Q \subseteq \bigcup M_j$. Thus $str \in \bigcup\{M_j \mid q_j \neq^\circ\}$. Therefore, $\exists M_j.str \in M_j$, hence $L_i \cap M_j \neq \emptyset$ and so $\Gamma \vdash S'_i <: T'_j$. ■

Lemma 38 *For all* Γ, S, T, L_Q, M_Q*, if:*

H1. $\Gamma \vdash S <: T$*,*

H2. $\Gamma \vdash L_Q \subseteq M_Q$*,*

H3. $inherit_\Gamma(S, L_Q) = S'$*, and*

H4. $inherit_\Gamma(T, M_Q) = T'$*,*

then $\Gamma \vdash S' <: T'$*.*

Proof: By double induction on syntactic size of S' and T', followed by case analysis of *inherit* in H3 and H4. We thus have four cases. In all cases, by inversion of (H1) and the definition of *inherit*, we have:

$$\begin{array}{rcl} S & = & \{L_1^{p_1} : S'_1 \cdots L_n^{p_n} : S'_n, L_A : \mathbf{abs}\} \\ T & = & \{M_1^{q_1} : T'_1 \cdots M_m^{q_m} : T'_m, M_A : \mathbf{abs}\} \end{array}$$

We thus have available the hypotheses of S-Object:

I1. $\forall i, j.L_i \cap M_j \neq \emptyset \Rightarrow p_i <: q_j \wedge \Gamma \vdash S'_i <: T'_j$,

I2. $\bigcup_i^{1\cdots m} M_i \subseteq \bigcup_j^{1\cdots n} L_j \cup L_A$,

I3. $M_A \subseteq L_A$,

I4. $\forall j.$if $q_j =\uparrow$ then $q_j =\uparrow \wedge \Gamma \vdash$ *inherit*$(S, M_j) <: T'_j$, and

I5. $\forall j.$if $M_j \cap L_A \neq \emptyset$ then $q_j =^\circ$ or $q_j =\uparrow$

Case 1. Base case, where "parent" of both S and T are Null or elided.

By definition of *inherit*, the goal is:

$$\Gamma \vdash \bigsqcup_{i}^{1\cdots n} \{S'_i \mid \Gamma \vdash L_Q \cap L_i \neq \emptyset\} <: \bigsqcup_{j}^{1\cdots m} \{T'_i \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset\}$$

The condition on both applications of *inherit* are $L_Q \subseteq \bigcup\{L_i \mid p_i \neq \circ\}$ and $M_Q \subseteq \bigcup\{M_j \mid q_j \neq \circ\}$. Therefore, $M_Q \subseteq \bigcup M_j$ and lemma 37 applies.

Case 2. Inductive case, where "parent" of both S and T are references to objects.

H5. $\exists L_i.\texttt{"parent"} \in L_i \wedge S'_i = \mathsf{Ref}\ S_P$,

H5'. $L_Q \subseteq \bigcup_i^{1\cdots n} L_i \cup L_A$,

H6. $\exists M_i.\texttt{"parent"} \in M_i \wedge T'_i = \mathsf{Ref}\ T_P$, and

H6'. $M_Q \subseteq \bigcup_i^{1\cdots m} M_i \cup M_A$.

HInd. $\forall L'_Q, M'_Q, S_P, T_P. |S_P| < |S| \wedge |T_P| < |T| \wedge \Gamma \vdash L'_Q \subseteq M'_Q \wedge \Gamma \vdash S_P <: T_P$ implies that $\left(\mathit{inherit}_\Gamma(S_P, L'_Q) = S'_P \wedge \mathit{inherit}_\Gamma(T_P, M'_Q) = T'_P \Rightarrow \Gamma \vdash S'_P <: T'_P\right)$.

The goal thus reduces to:

$$\begin{aligned}\Gamma \quad \vdash \quad & \bigsqcup\{S'_i \mid \Gamma \vdash L_Q \cap L_i \neq \emptyset\} \sqcup \mathit{inherit}(S_p, \mathcal{L}) <: \bigsqcup\{T'_i \mid \Gamma \vdash M_Q \cap M_i \neq \emptyset\} \sqcup \mathit{inherit}(T_p, \mathcal{M}) \\ & \text{where } \mathcal{L} = L_Q \cap (L_A \cup \bigcup\{L_i | p_i = \circ\}) \text{ and } \mathcal{M} = M_Q \cap (M_A \cup \bigcup\{M_j | q_j = \circ\})\end{aligned}$$

We define the set of inherited fields of T that are looked up by M_Q and are absent on S:

$$\begin{aligned}\mathcal{N} &= \{M_j \mid M_Q \cap M_j \cap L_A \neq \emptyset \wedge q_j = \uparrow\} \\ \mathcal{L}^+ &= \mathcal{L} \cap \bigcup \mathcal{N} \\ \mathcal{L}^- &= \mathcal{L} \cap \overline{\bigcup \mathcal{N}}\end{aligned}$$

Using lemma 36, we rewrite the goal to:

$$\begin{aligned}\Gamma \quad \vdash \quad & \bigsqcup\{S'_i \mid \Gamma \vdash L_Q \cap L_i \neq \emptyset\} \sqcup \mathit{inherit}(S_p, \mathcal{L} \cap \overline{\bigcup \mathcal{N}}) \sqcup \mathit{inherit}(S_p, \mathcal{L} \cap \bigcup \mathcal{N}) \\ <: \quad & \bigsqcup\{T'_i \mid \Gamma \vdash M_Q \cap M_i \neq \emptyset\} \sqcup \mathit{inherit}(T_p, \mathcal{M})\end{aligned}$$

We prove the goal by breaking it into the following subcases:

a. $\Gamma \vdash \bigsqcup\{S'_i \mid \Gamma \vdash L_Q \cap L_i \neq \emptyset\} <: \bigsqcup\{T'_j \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset\}$

We cannot apply lemma 37 directly because $M_Q \subseteq \bigcup M_j \cup M_A$, whereas the hypothesis of the lemma requires $M_Q \subseteq \bigcup M_j$.

However, note that since $L_A \cap L_i = \emptyset$ (by well-formedness of types), we have $L_Q \cap L_i \neq \emptyset$ iff $L_Q \backslash L_A \cap L_i$. Similarly, $M_Q \cap M_j \neq \emptyset$ iff $M_Q \backslash M_A \cap M_J$. We can therefore rewrite the subgoal to $\Gamma \vdash \bigsqcup\{S'_i \mid \Gamma \vdash L_Q \backslash L_A \cap L_i \neq \emptyset\} <: \bigsqcup\{T'_j \mid \Gamma \vdash M_Q \backslash M_A \cap M_j \neq \emptyset\}$. Lemma 37 now applies.

b. $\Gamma \vdash \mathit{inherit}(S_p, \mathcal{L} \cap \overline{\bigcup \mathcal{N}}) <: \mathit{inherit}(T_p, \mathcal{M})$

By induction (HInd). The following cases allow us to apply HInd.

– $|S_P| < |S|$ and $|T_P| < |T|$ are trivial.

– We show that $\Gamma \vdash S_P <: T_P$. By H5 and H6, `"parent"` $\in L_P, M_P$ thus $L_P \cap M_P \neq \emptyset$. Therefore, $\Gamma \vdash \mathsf{Ref}\ S_P <: \mathsf{Ref}\ T_P$ by I1. By (S-Ref), it follows that $\Gamma \vdash S_P <: T_P$.

– We show that $\Gamma \vdash \mathcal{L} \cap \overline{\bigcup \mathcal{N}} \subseteq \mathcal{M}$. It is sufficient to show that $\forall x. \Gamma \vdash x \in \mathcal{L} \cap \overline{\bigcup \mathcal{N}} \Rightarrow \Gamma \vdash x \in \mathcal{M}$.

By definition of $\mathcal{L}$, $x \in L_Q$, thus by (H2), $x \in M_Q$.

By (H6'), $x \in M_A \cup \bigcup_j^{j \cdots m} M_j$. If $x \in M_A$, we are done. So, consider the case where $\exists j. x \in M_j$.

By definition of $\mathcal{L}$, either $x \in L_A$ or $x \in \bigcup\{L_i | p_i = \circ\}$. If $x \in L_A$, then by I5 $q_j = \circ$. If $x \in \bigcup\{L_i | p_i = \circ\}$, since $x \in L_i \cap M_j$, $p_i <: q_j$, and by p-Refl, $q_j = \circ$.

Therefore, $x \in \mathcal{M}$ since it is in both sets.

c. $\Gamma \vdash \mathit{inherit}(S_p, \mathcal{L} \cap \bigcup \mathcal{N}) <: \bigsqcup\{T'_j \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset\}$.

Rewrite the left-hand side by expanding the definition of $\mathcal{N}$, distributing the intersection over the union, and applying lemma 36:

$$\Gamma \vdash \bigsqcup\{\mathit{inherit}(S_p, \mathcal{L} \cap M_j) \mid M_Q \cap M_j \cap L_A \neq \emptyset \wedge q_j = \uparrow\} <: \bigsqcup\{T'_j \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset\}$$

It is sufficient to show that for all elements of the left-hand side, there exists a supertype on the right-hand side. For each $\mathit{inherit}(S_p, \mathcal{L} \cap M_j)$ on the left-hand side, the associated T'_j is on the right-hand side by definition of $\mathcal{N}$. We now show that

$$\Gamma \vdash \mathit{inherit}(S_p, \mathcal{L} \cap M_j) <: T'_j$$

Since $M_j \cap L_A \neq \emptyset$ and $q_j = \uparrow$, I4 applies and $\Gamma \vdash \mathit{inherit}(S, M_j) <: T'_j$. By lemma 36 $\mathit{inherit}(S, M_j) = \mathit{inherit}(S, \mathcal{L} \cap M_j) \sqcup \mathit{inherit}(S, \overline{\mathcal{L}} \cap M_j)$, so $\Gamma \vdash \mathit{inherit}(S, \mathcal{L} \cap M_j) <: T_j$ by the definition of joins. Further, by the definition of *inherit*,

$$\Gamma \vdash \mathit{inherit}(S, \mathcal{L} \cap M_j) = \mathit{inherit}(S_P, \mathcal{L} \cap M_j \cap L_A \bigcup \{L_i \mid p_i = \circ\}) \sqcup \ldots$$

and by the definition of $\mathcal{L}$, $\mathcal{L} \cap M_j \cap L_A \bigcup \{L_i \mid p_i = \circ\}$ is the same as $\mathcal{L} \cap M_j$. By the definition of joins:

$$\Gamma \vdash \mathit{inherit}_\Gamma(S_P, \mathcal{L} \cap M_j) <: T_j$$

which, when applied for each j, completes Case 2.

Case 3. Impossible case, where the "parent" field is on the right-hand side type, but is elided on the left-hand side type.

By well-formedness of types, if $\exists i.$"parent" $\in M_i$ then $q_i =^\downarrow$. But by I2, "parent" $\in \bigcup_j^{1\cdots n} L_j \cup L_A$, which is a contradiction.

Case 4. Inductive case, where the "parent" field is on the left-hand side type, but is elided or null on the right-hand side type.

HLP. $\exists L_P.$`"parent"` $\in L_P \wedge S'_i = \mathsf{Ref}\ S_P$,

H6. $L_Q \subseteq \bigcup_i^{1\cdots n} L_i \cup L_A$,

HRP. If $\exists M_P.$`"parent"` $\in M_P$ then $T'_P = \mathsf{Null}$, and

H8. $M_Q \subseteq \bigcup_i^{1\cdots m} \{M_i \mid q_i \neq^\circ\}$.

There are two cases of HRP. First, if $\exists M_P.$`"parent"` $\in M_P$ holds, then $L_P \cap M_P \neq \emptyset$. Thus by I1, it must be that $\Gamma \vdash \mathsf{Ref}\ S_P <: \mathsf{Null}$, which is a contradiction. Therefore, we consider the second case, where $\neg\exists M_P.$`"parent"` $\in M_P$. The goal is therefore:

$$\bigsqcup_i^{1\cdots n} \{S'_i \mid \Gamma \vdash L_Q \cap L_i \neq \emptyset\} \sqcup \mathit{inherit}(S_P, L_Q \cap (L_A \cup \bigcup \{L_i | p_i = \circ\})) <: \bigsqcup_j^{1\cdots m} \{T'_j \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset\}$$

The first part of the join is satisfied by lemma 37. For part 2, using lemma 36 rewrite:

$$\Gamma \vdash \mathit{inherit}(S_P, L_Q \cap (L_A \cup \bigcup\{L_i | p_i = \circ\})) <: \bigsqcup_{j}^{1\cdots m} \{T'_j \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset\}$$

to:

$$\Gamma \vdash \mathit{inherit}(S_P, L_Q \cap L_A) \sqcup \mathit{inherit}(S_P, L_Q \cap \bigcup\{L_i | p_i = \circ\}) <: \bigsqcup_{j}^{1\cdots m} \{T'_j \mid \Gamma \vdash M_Q \cap M_j \neq \emptyset\}$$

There are two cases.

- Consider *str* $\in L_Q \cap L_A$. This case follows by the same argument as in item c. of Case 2.
- Consider *str* $\in L_Q \cap L_i$, where $p_i = \circ$. By H2, *str* $\in M_Q$. By H8, there exists an M_j, such that *str* $\in M_j$ and $q_j \neq^\circ$. By I1, since *str* $\in L_i, M_j$, it must be that $p_i <: q_j$. By inspection of the definition of the $p <: q$ relation, we have a contradiction.

■

Lemma 39 *If* $\{L_1 : F_1 \cdots L_n : F_n\} <: \{M_1 : G_1 \cdots M_m : G_m\}$*, then* $\bigcup_j^{1\cdots m}\{M_j \mid G_j = T^{\downarrow}\} \subseteq \bigcup_i^{1\cdots n}\{L_i \mid F_i = T^{\downarrow}\}$.

Proof: By contradiction.

Assume there exists some string x with $x \in \bigcup_j^{1\cdots j}\{M_j \mid G_j = T^{\downarrow}\}$ and $x \notin \bigcup_i^{1\cdots n}\{L_i \mid F_i = T^{\downarrow}\}$. By assumption of S-Object, $\bigcup_j^{1\cdots m} M_j \subseteq \bigcup_i^{1\cdots n} L_i$, so it must be that there is some L_i that contains x, but either has type T° or **abs**. That is, there must be an M_j with $x \in M_j$ and an L_i with $x \in L_i$ with $G_j = T^{\downarrow}$ and either $F_i =$ **abs** or $F_i = T'^\circ$. This violates S-Object, which asserts that since $L_i \cap M_j \neq \emptyset$, it must be that $F_i <: G_j$, which cannot happen since possibly absent and definitely absent fields cannot subtype definitely present fields.

■

D.4 Typing

Lemma 40 (Canonical Forms) *If* $\Sigma; \Gamma \vdash v : T$ *and if* T *is:*

$$\frac{\begin{array}{c}\Gamma \vdash v : S \cdots \qquad \Gamma(f) = L \qquad \Sigma; \Gamma, \alpha <: L, f : \alpha \vdash e_2 : T \\ L' = L \cap \overline{\alpha} \qquad \Gamma \vdash e_3 : T\end{array}}{\Sigma; \Gamma \vdash \textsf{\textbf{if}}\ (\{str : v \cdots\}\ \textsf{\textbf{hasfield}}\ f)\ e_2\ \textsf{\textbf{else}}\ e_3 : T} \qquad (\text{T-IfHasField1})$$

$$\frac{\begin{array}{c}\Gamma(o) = \{\cdots L^{\circ} : S \cdots\} \qquad \Sigma; \Gamma, o : \{\cdots str^{\downarrow} : S, L'^{\circ} : S \cdots\} \vdash e_2 : T \\ L' = L \cap \overline{\{str\}} \qquad \Gamma \vdash e_3 : T\end{array}}{\Sigma; \Gamma \vdash \textsf{\textbf{if}}\ (o\ \textsf{\textbf{hasfield}}\ str)\ e_2\ \textsf{\textbf{else}}\ e_3 : T} \qquad (\text{T-IfHasField2})$$

$$\frac{\Gamma \vdash v : S \cdots \qquad \Sigma; \Gamma \vdash e_3 : T \qquad str_2 \notin str \cdots}{\Sigma; \Gamma \vdash \textsf{\textbf{if}}\ (\{str : v \cdots\}\ \textsf{\textbf{hasfield}}\ str_2)\ e_2\ \textsf{\textbf{else}}\ e_3 : T} \qquad (\text{T-IfHasFieldFalse})$$

$$\frac{\Sigma; \Gamma \vdash e_3 : T}{\Sigma; \Gamma \vdash \textsf{\textbf{if}}\ (\textsf{\textbf{false}})\ e_2\ \textsf{\textbf{else}}\ e_3 : T} \qquad (\text{T-IfFalse})$$

Figure D.1: Auxiliary Typing Rules for If-Splitting

$$\begin{tikzcd}
 & \text{T-IfHasField} \arrow[ld, "{[o/v]}"'] \arrow[rd, "{[f/v]}"] & \\
\text{T-IfHasField1} \arrow[d] \arrow[rrdd] & & \text{T-HasField2} \arrow[lld] \arrow[dd] \\
\text{T-IfHasFieldFalse} \arrow[d] & & \\
\text{T-IfFalse} & & \text{T-If}
\end{tikzcd}$$

Figure D.2: Usage of Auxiliary Typing Rules by Substitution

- $\{L_1^{p_1} : S_1 \cdots L_m^{p_m} : S_m\}$ *then* $v =$ `{`str_1`:`$v_1 \cdots$`}`, $\Sigma;\Gamma \vdash v_1 \cdots v_n : U_1 \cdots U_n$, $\Sigma;\Gamma \vdash v : \{str_1^{\downarrow} : U_1 \cdots str_n^{\downarrow} : U_n, \overline{\{str_1 \cdots str_n\}} : \mathbf{abs}\}$ *and* $\{str_1^{\downarrow} : U_1 \cdots str_n^{\downarrow} : U_n, \overline{\{str_1 \cdots str_n\}} : \mathbf{abs}\} <: \{L_1^{p_m} : S_1 \cdots L_m^{p_m} : S_m\}$,

- $\textsf{Ref}\, S$, *then* $v = loc$ *and* $\Sigma(loc) <: S$,

- $S \to T$, *then* $v =$ **func**`(`x`) {` e `}`,

- L *then* $v = str$ *and* $\Gamma \vdash str <: L$

Proof: By induction on the typing derivation.

Lemma 41 (Inversion) *If:*

- $\Sigma;\Gamma \vdash$ `{`str`:`$v \cdots$`}` $: T$ *then* $\Sigma;\Gamma \vdash v : S \cdots$ *and* $\Gamma \vdash \{str^{\downarrow} : S \cdots\} <: T$

- $\Sigma;\Gamma \vdash$ **ref** $e : T$ *then* $\Sigma;\Gamma \vdash e : S$ *and* $\textsf{Ref}\ S = T$

- $\Sigma;\Gamma \vdash l : T$ *then* $\Sigma(l) = T$

- $\Sigma;\Gamma \vdash$ **deref** $e : T$, *then* $\Sigma;\Gamma \vdash e : \textsf{Ref}\, S$ *with* $S <: T$,

- $\Sigma;\Gamma \vdash e_1$ `=` $e_2 : T$, *then* $\Sigma;\Gamma \vdash e_1 : \textsf{Ref}\, S$, $\Sigma;\Gamma \vdash e_2 : U$, $U <: S$, *and* $\textsf{Ref}\, S <: T$,

- $\Sigma;\Gamma \vdash e_f$`(`$e \cdots$`)` $: T$, *then* $\Sigma;\Gamma \vdash e_f : S \cdots \to T'$, $\Sigma;\Gamma \vdash e : S \cdots$, *and* $T' <: T$.

- $\Sigma;\Gamma \vdash e_o$`[`e_f`]` $: T$, *then* $\Sigma;\Gamma \vdash e_o : \{L_1^{p_1} : S_1 \cdots\}$, $\Sigma;\Gamma \vdash e_f : L$, *inherit*$(\{L_1^{p_1} : S_1 \cdots\}, L) = T'$, *and* $T' <: T$.

- $\Sigma;\Gamma \vdash e_o$`[`e_f `=` e_v`]` $: T$ *then* $\Sigma;\Gamma \vdash e_o : \{L^p : S \cdots, L_A : \mathbf{abs}\}$, $\Sigma;\Gamma \vdash e_f : L'$, $\Sigma;\Gamma \vdash e_v : U$, $\forall L.$*if* $L \cap L' \neq \emptyset$ *then* $U <: S$, *and* $\Gamma \vdash \{L^p : S \cdots, L_A : \mathbf{abs}\} <: T$.

- $\Sigma;\Gamma \vdash$ **delete** e_o`[`e_f`]` $: T$, *then* $\Sigma;\Gamma \vdash e_o : \{L_1^{p_1} : S_1 \cdots\}$, $\Sigma;\Gamma \vdash e_f : L$, $\forall L \cap L_i \neq \emptyset. F_i \neq T^{\downarrow}$, $\Gamma \vdash \{L_1^{p_1} : S_1 \cdots\} <: T$

- $\Sigma;\Gamma \vdash e_1$ **hasfield** $e_2 : \textsf{Bool}$, *then* $\Sigma;\Gamma \vdash e_1 : \{L_1 : F_1 \cdots L_n : F_n\}$, $\Sigma;\Gamma \vdash e_2 : L$.

- $\Sigma;\Gamma \vdash e$ **matches** $P : \textsf{Bool}$, *then* $\Sigma;\Gamma \vdash e : L$.

- $\Sigma;\Gamma \vdash$ **if** `(`v_1`) {` e_2 `}` **else** `{` e_3 `}` $: T$, *then* $\Sigma;\Gamma \vdash v_1 : \textsf{Bool}$, $\Sigma;\Gamma \vdash e_2 : T$, *and* $\Sigma;\Gamma \vdash e_3 : T$.

- $\Sigma; \Gamma \vdash$ **fieldin** v_{obj} *init* v_{acc} **do** $v_f : T$, *then* $\Sigma; \Gamma \vdash v_{obj} : \{L^p : S \cdots\}$, $\Sigma; \Gamma \vdash v_{acc} : T$, *and* $\Sigma; \Gamma \vdash v_f : (\mathit{Str} \to T) \to T$

Lemma 42 (Type Substitution) *If* $\Sigma; \alpha <: S, \Gamma \vdash e : T$ *and* $\Gamma \vdash U <: S$ *then* $\Sigma; \Gamma[\alpha/U] \vdash e[\alpha/U] : T[\alpha/U]$.

Proof: By induction on the typing derivation. ■

Lemma 43 (Substitution) *If* $\Sigma; x : S, \Gamma \vdash e : T$ *and* $\Sigma; \Gamma \vdash v : S$, *then* $\Sigma; \Gamma \vdash e[x/v] : T$.

Proof: By induction on the typing derivation. The only interesting case is substituting the identifiers in **if** (o **hasfield** f) e_2 **else** e_3 when it is typed by T-IfHasField. The resulting expressions require the auxiliary typing rules in fig. D.1.

- The expression is typed by T-HasField and $x = o$. The resulting expression is typable by T-IfHasField1 as follows. By canonical forms, $v = \{str : v \cdots\}$ and $\Sigma; \Gamma \vdash v' : T'$. By induction, $\Sigma; \Gamma, \alpha <: L, f : \alpha e_2[x/v]$ and $\Sigma; \Gamma \vdash e_3[x/v]$. The remaining antecents of T-IfHasField1 are those of T-HasField.

- The expression is typed by T-HasField and $x = f$. The resulting expression is typable by T-IfHasField2 as follows. By canonical forms, $v = str$, $\Sigma; \Gamma \vdash v : str$, and $\Gamma \vdash str <: L$. By type substitution followed by induction, $\Sigma; \Gamma, o : \{\cdots str^{\downarrow} : S, L'^{\circ} : S \cdots\} \vdash e_2 : T$. The remaining antecedents of T-IfHasField2 are those of T-HasField.

- The expression is typed by T-IfHasField1 and $x = f$. The resulting expression has the form:

  if ({ str: $v' \cdots$ } **hasfield** str') e_2 **else** e_3

  There are two cases.

  – If $str' \in str \cdots$ then by type substitution and induction, $\Sigma; \Gamma, \alpha <: L, f : \alpha \vdash e_2 : T[\alpha/str][f/v] = \Sigma; \Gamma \vdash e_2[f/v] : T$. By induction, $\Sigma; \Gamma \vdash e_3[f/v] : T$. Finally, the conditional has type Bool. Thus the expression is typable by T-If.

  – If $str' \notin str \cdots$ then the term is trivally typable by T-IfHasFieldFalse.

- The expression is typed by T-IfHasField2 and $x = o$. The resulting expression has the form:

  if ({ str: $v' \cdots$ } **hasfield** str') e_2 **else** e_3

There are two cases.

– If $str' \in str \cdots$ then the result is trivially typable by T-If.

– If $str' \notin str \cdots$ then the term is trivally typable by T-IfHasFieldFalse.

■

Lemma 44 (Main Preservation) *If $\Sigma_1; \cdot \vdash ae : T$, $\Sigma_1 \vdash \sigma_1$, and $\sigma_1 E\langle ae\rangle \rightarrow \sigma_2 E\langle e_2\rangle$ then there exists a Σ_2, such that:*

i. $\Sigma_2 \supseteq \Sigma_1$,

ii. $\Sigma_2 \vdash \sigma_2$, *and*

iii. $\Sigma_2; \cdot \vdash e_2 : T$.

Proof: By case-analysis on ae, using inversion (lemma 41) where specified:

- $\sigma_1 E\langle(\textsf{func } (x : S') \texttt{ \{ } e \texttt{ \}})(v)\rangle \rightarrow \sigma_1 E\langle e[x/v]\rangle$.

  By inversion, $\Sigma_1; \cdot \vdash v : S$, $\cdot \vdash S <: S'$, $\Sigma_1; x : S \vdash e : T'$, and $\cdot \vdash T' <: T$. By substitution (lemma 43), $\Sigma_1; \cdot \vdash e[x/v] : T$.

- $\sigma E\langle(\lambda\alpha <: S.e)(U)\rangle \rightarrow \sigma E\langle e[\alpha/U]\rangle$

  By type substitution (lemma 42).

- $\sigma E\langle \textsf{ref } v\rangle \rightarrow \sigma, (l, v)E\langle l\rangle$ where $l \notin dom(\sigma)$. By inversion, $\Sigma_1; \cdot \vdash v : S$ and $\textsf{Ref } S <: T$. Let $\Sigma_2 = l : S, \Sigma_1$. By T-Loc, $\Sigma_2; \cdot \vdash l : \textsf{Ref } S$.

- $\sigma E\langle \textsf{deref l}\rangle \rightarrow \sigma E\langle \sigma(l)\rangle$

  By inversion, $\Sigma; \Gamma \vdash l : \textsf{Ref } S$ and $\Gamma \vdash S <: T$. By inversion, $\Sigma(l) = S$. Thus by T-Loc and T-Sub $\Sigma; \Gamma \vdash l : T$.

- $\sigma E\langle \textsf{setref } l \ v\rangle \rightarrow \sigma[l := v]E\langle l\rangle$ where $l \in dom(\sigma)$.

  By inversion (T-SetRef), $\Sigma; \Gamma \vdash l : \textsf{Ref } \ S$, $\Sigma; \Gamma \vdash v : S$, and $\textsf{Ref } \ S = T$. By inversion (T-Loc), $\Sigma(l) = T$ thus $\Sigma \vdash \sigma[l := v]$. by T-Loc, $\Sigma; \Gamma \vdash l : T$.

- $\sigma E\langle\texttt{\{ }\cdots str\texttt{: } v\cdots\texttt{ \}[}str\texttt{]}\rangle \rightarrow \sigma E\langle v\rangle$. By inversion, $\Sigma_1;\cdot\vdash str : L$, $\Sigma_1;\cdot\vdash\{\cdots str_1 : v_1\cdots\} : S$, $\Sigma_1;\cdot\vdash v : T'$, $T' <: \mathit{inherit}.(S, L)$, and $\mathit{inherit}.(S, L) <: T$. By inversion, $\{\cdots str^{\downarrow} : T'\cdots\} <: S$. Thus $\cdot\vdash T' <: T$.

  By lemma 38, $\mathit{inherit}.(\{\cdots str^{\downarrow} : T'\cdots\}, str) <: \mathit{inherit}.(S, L)$, and finally $\mathit{inherit}.(\{\cdots str^{\downarrow} : T'\cdots\}, str) = T'$.

- $\sigma E\langle\texttt{\{ }\cdots\texttt{ "parent": } l\texttt{ \}[}str\texttt{]}\rangle \rightarrow \sigma E\langle\texttt{(}\textbf{deref}\ l\texttt{)[}str\texttt{]}\rangle$, where $str \notin \cdots$.

  By inversion of the LHS, $\Sigma_1;\cdot\vdash str : L$, $\Sigma_1;\cdot\vdash\{\cdots\text{"parent"} : T_P\} : S$, and $\mathit{inherit}.(S, L) <: T$. By lemma 38, $\mathit{inherit}.(\{\cdots\text{"parent"}^{\downarrow} : T_P\}, str) <: \mathit{inherit}.(S, L)$. By inversion, $\Sigma_1;\cdot\vdash l : \mathsf{Ref}\ S_P = T_P$. Since $str \notin \cdots$ and by definition of $\mathit{inherit}$, $\mathit{inherit}.(\{\cdots\text{"parent"}^{\downarrow} : \mathsf{Ref}\ S_P\}, str) = \mathit{inherit}_{\ldots}(S_P, str)$, which is a subtype of T.

  Type right-hand side with T-Sub and T-GetField, using $\Sigma_1;\cdot\vdash str : str$, $\mathit{inherit}_{\ldots}(S_P, str) <: T$, and $\Sigma_1;\cdot\vdash\texttt{(}\textbf{deref}\ l\texttt{)}: S_P$. This holds since $\Sigma_1 : \cdot\vdash l : \mathsf{Ref}\ S_P$ above.

- $\sigma E\langle\texttt{\{ }\cdots str : v\cdots\texttt{ \}[}str\texttt{ = }v'\texttt{]}\rangle \rightarrow \sigma E\langle\texttt{\{ }\cdots str : v'\cdots\texttt{ \}}\rangle$

  By inversion (T-Update), $\Sigma;\Gamma\vdash\texttt{\{}\cdots\texttt{\}} : \{L : S\cdots\}$, $\Gamma\vdash\{L : S\cdots\} <: T$, and $\Sigma;\Gamma\vdash v' : U'$. By inversion (T-Object), $\Sigma;\Gamma\vdash\texttt{\{}\cdots str\texttt{:}v\cdots\texttt{\}} : \{\cdots str : U\cdots\}$ and $\Gamma\vdash\{\cdots str : U\cdots\} <: \{L : S\cdots\}$. Thus by inversion (T-Update), $\Gamma\vdash U' <: U$. The resulting expression is typable by S-Object, thus by S-Sub, $\Gamma\vdash\{\cdots str : U'\cdots\} <: \{\cdots str : U\cdots\} <: T$.

- $\sigma E\langle\texttt{\{ }\cdots\texttt{ \}[}str\texttt{ = }v'\texttt{]}\rangle \rightarrow \sigma E\langle\texttt{\{ }\cdots\texttt{ \}}\rangle$ where $str \notin \cdots$

  By inversion of T-Update, $\Sigma;\Gamma\vdash\texttt{\{}\cdots str\texttt{:}v\cdots\texttt{\}} : S$ and $\Gamma\vdash S <: T$.

- $\sigma E\langle\textbf{delete}\texttt{ \{ }\cdots str : v\cdots\texttt{ \}[}str\texttt{]}\rangle \rightarrow \sigma E\langle\texttt{\{ }\cdots\cdots\texttt{ \}}\rangle$

  Similar to to E-UpdateField case above.

- $\sigma E\langle\textbf{delete}\texttt{ \{ }\cdots\texttt{ \}[}str\texttt{]}\rangle \rightarrow \sigma E\langle\textbf{delete}\texttt{ \{ }\cdots\texttt{ \}}\rangle$ where $str \notin \cdots$

  Similar to to E-UpdateField case above.

- $\sigma E\big\langle\textbf{fieldin}\texttt{ \{ }s\texttt{:}v\texttt{,}rest\cdots\texttt{ \} init }v_{acc}\textbf{ do }v_f\big\rangle \rightarrow \sigma E\big\langle\textbf{fieldin}\texttt{ \{ }str_2 : v_2\ \cdots\texttt{ \} init }v_f\texttt{(}str_1\texttt{)(}v_{acc}\texttt{)}\textbf{ do }v_f\big\rangle$

  By inversion, $\Sigma;\Gamma\vdash v_{acc} : T$ and $\Sigma;\Gamma\vdash v_f : (\mathsf{Str}\rightarrow T)\rightarrow T$. The double application can by typed by T-App, and the resulting expression will by typable by T-FieldIn.

- $\sigma E\langle$**fieldin** `{` s:v`}` init v_{acc} **do** $v_f\rangle \rightarrow \sigma E\langle v_f$`(`$s$`)(`$v_{acc}$`)`$\rangle$

  Similar to E-FieldIn above.

  The remaining cases are conventional and straightforward. (**if** is standard, and in the rest, both the left-hand side and the right-hand side have type Bool.)

- $\sigma E\langle$**if**`(`**true**`)` `{` e_1 `}` **else** `{` e_2 `}`$\rangle \rightarrow \sigma E\langle e_1\rangle$

- $\sigma E\langle$**if**`(`**false**`)` `{` e_1 `}` **else** `{` e_2 `}`$\rangle \rightarrow \sigma E\langle e_2\rangle$

- $\sigma E\langle$`{` $\cdots str$:$v\cdots$ `}` **hasfield** $str\rangle \rightarrow \sigma E\langle$**true**$\rangle$

- $\sigma E\langle$`{` $\cdots$ `}` **hasfield** $str\rangle \rightarrow \sigma E\langle$**false**$\rangle$ where $str \notin \cdots$

- $\sigma E\langle str$ **matches** $P\rangle \rightarrow \sigma E\langle$**true**$\rangle$

- $\sigma E\langle str$ **matches** $P\rangle \rightarrow \sigma E\langle$**false**$\rangle$

■

Theorem 10 (Preservation) *If $\Sigma_1 \vdash e_1 : T$, $\Sigma_1 \vdash \sigma_1$, and $\sigma_1 e_1 \rightarrow \sigma_1 e_2$, then there exists a Σ_2, such that:*

i. $\Sigma_2; \cdot \vdash e_2 : T$,

ii. $\Sigma_2 \vdash \sigma_2$, *and*

iii. $\Sigma_1 \subseteq \Sigma_2$.

Proof: By case-analysis of the reduction rules, there exists an evaluation context, E, an active expression, ae, and an expression, e', such that $e_1 = E\langle ae\rangle$ and $e_2 = E\langle e'\rangle$. There thus exists a subdeduction $\Sigma_1; \cdot \vdash ae : S$ of the original typing derivation. Lemma 44 now applies, so we have $\Sigma_2 \subseteq \Sigma_1$, $\Sigma_2 \vdash \sigma_2$, and $\Sigma_2; \cdot \vdash e' : S$. Replacing the original subdeduction, we have $\Sigma_2; \cdot \vdash E\langle e'\rangle : T$. ■

Theorem 11 (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 $\sigma e \rightarrow \sigma' e'$.*

Proof: By case-analysis of the reduction rules, either $e \in v$, $e = E\langle ae \rangle$, or $e = E\langle \textsf{err} \rangle$. By inspection of the typing relation, **err** is untypable. We therefore consider the case where $e = E\langle ae \rangle$ by case-analysis on the definition of active expressions.

- The cases where ae is of the form $v_1\texttt{(}v_2\texttt{)}$, **ref** v, **deref** v, v_1 `=` v_2, and **if** `(`v_1`) {` e_2 `}` **else** `{` e_3 `}` are routine.

- Consider $ae = v_1\texttt{[}v_2\texttt{]}$. By inversion, $\Sigma; \cdot \vdash v_1 : \{L_1^{p_1} : T_1 \cdots L_n^{p_n} : T_n\}$ and $\Sigma; \cdot \vdash v_2 : L_Q$. By canonical forms, $v_1 = \{str_1 : w_1 \cdots str_m : w_m\}$ and $\{str_1^{\downarrow} : S_1 \cdots str_m^{\downarrow} : S_m, \overline{\{str_1 \cdots str_m\}}\} <: \{L_1^{p_1} : T_1 \cdots L_n^{p_n} : T_n\}$. Also by canonical forms, $v_2 = str_Q$ and $str_Q <: L_Q$.

  Therefore, we must only demonstrate that the E-NotFound reduction is not applicable in the case where "parent" is **null**. It is sufficient to show that $str_Q \in \{str_1 \cdots str_m\}$, which we do by the following sequence of inclusions:

  By S-Str, $str_Q \in L_Q$. By definition of *inherit*, when "parent" is **null**, $L_Q \subseteq \bigcup\{L_i \mid p_i \neq^\circ\}$. Evidently, $\bigcup\{L_i \mid p_i \neq^\circ\} \subseteq \bigcup L_i$. By antecedent (2) of S-Object, we have that $\bigcup L_i \subseteq \{str_1 \cdots str_m\}$.

- Consider $ae = v_1\texttt{[}v_2 \texttt{ = } v_3\texttt{]}$. By inversion, $\Sigma; \cdot \vdash v_1 : \{L^p : S \cdots\}$ and $\Sigma; \cdot \vdash v_2 : L_Q$. By canonical forms, $v_1 = \{str : w \cdots\}$ and $v_2 = str_Q$. Thus either E-Create or E-Update apply.

- Consider $ae =$ **delete** $v_1\texttt{[}v_2\texttt{]}$. By inversion, $\Sigma; \cdot \vdash v_1 : \{L^p : S \cdots\}$ and $\Sigma; \cdot \vdash v_2 : L_Q$. By canonical forms, $v_1 = \{str : w \cdots\}$ and $v_2 = str_Q$. Thus either E-Delete or E-Delete-None apply.

- Consider $ae = v_1$ **hasfield** v_2. By inversion, $\Sigma; \cdot \vdash v_1 : \{L^p : S \cdots\}$ and $\Sigma; \cdot \vdash v_2 : L_Q$. By canonical forms, $v_1 = \{str : w \cdots\}$ and $v_2 = str_Q$. Thus either E-HasField or E-HasNotField apply.

- Consider $ae = v$ **matches** P. By inversion and canonical forms, $v = str$. Thus either E-Matches or E-NoMatch apply.

- Consider $ae =$ **fieldin** `{` $s_1 : v_1$`,` $s_2 : v_2$ $\cdots$`}` init v_{acc} **do** v_f. By inversion, $\Sigma; \cdot \vdash v_1 : \{L^p : S \cdots\}$. By canonical forms, $v_1 = \{str : w \cdots\}$. Thus either E-FieldIn or E-FieldIn-End apply.

Appendix E

The ADsafe Environment

This appendix contains the type environment we use to type-check ADsafe. Widget is the recursively type of values that untrusted widgets can manipulate. In the concrete syntax below, the field name $*$ is shorthand for all fields, except those explicitly marked BAD or given a type.

```
type Widget = trec t .
    Undef
  + Null
  + Num
  + Int
  + Str
  + Bool
  + { "prototype" : BAD,
      ___nodes___ : Array<HTMLElement + Undef> + Undef,
      ___star___ : Bool + Undef,
      __proto__ : BAD,
      __parent__ : BAD,
      valueOf : _,
      hasOwnProperty : _,
      toString : _,
      arguments : BAD,
      "constructor" : BAD,
      watch : BAD,
      unwatch : BAD,
      #proto : Object+Function+Array+Bunch_proto+Str+Num+Bool+RegExp,
      * : 't,
      #code : ['t + HTMLWindow] 't ... -> 't
     }
```

The WidgetObj type is the object portion of the Widget type. It makes it easier to write some types below.

```
type WidgetObj =
    { "prototype" : BAD,
      ___nodes___ : Array<HTMLElement + Undef> + Undef,
      ___star___ : Bool + Undef,
      __proto__ : BAD,
      __parent__ : BAD,
      valueOf : _,
      hasOwnProperty : _,
      toString : _,
      arguments : BAD,
      "constructor" : BAD,
      watch : BAD,
      unwatch : BAD,
      #proto : Object+Function+Array+Bunch_proto+Str+Num+Bool+RegExp,
      * : 'Widget,
      #code : ['Widget + HTMLWindow] 'Widget ... -> 'Widget
     }
```

Selector is the type we ascribe to CSS selectors for ADsafe. CSS selectors have several other fields, but these are the only ones that ADsafe requires.

```
type Selector =
    { op: Str + Undef,
      name: Str + Undef,
      value: Str + Undef,
      #proto: Object,
      *: Bot,
      #code: Bot }
```

The reject_name function is a key check in ADsafe that returns **true** on strings in banned and **false** otherwise.

```
type not_banned =
     $^{"arguments", "caller", "callee", "constructor", "eval", "stack",
        "watch", "unwatch", "valueOf", "toString", "hasOwnProperty",
        "prototype", "___nodes___", "___star___", "__proto__", "__parent__",
        "__defineGetter__", "__defineSetter__"}

type banned =
     ${"arguments", "caller", "callee", "constructor", "eval", "stack",
       "watch", "unwatch", "valueOf", "toString", "hasOwnProperty",
       "prototype", "___nodes___", "___star___", "__proto__", "__parent__",
        "__defineGetter__", "__defineSetter__"} +
     Bool + Null + Undef + 'WidgetObj + Num

reject_name : (('banned -> True) + ('not_banned -> Bool))
```

A Pecker is an ADsafe abstraction for selecting elements.

```
type Pecker =
     {".":   HTMLElement + Undef -> Bool ,
      "&":   HTMLElement + Undef -> Bool ,
      "_":   HTMLElement + Undef -> Bool ,
      "[":   HTMLElement + Undef -> Bool ,
      "[=":  HTMLElement + Undef -> Bool ,
      "[!=": HTMLElement + Undef -> Bool ,
      "[^=": HTMLElement + Undef -> Bool ,
      "[$=": HTMLElement + Undef -> Bool ,
      "[*=": HTMLElement + Undef -> Bool ,
      "[~=": HTMLElement + Undef -> Bool ,
      "[|=": HTMLElement + Undef -> Bool ,
      ":blur": HTMLElement + Undef -> Bool ,
      ":checked": HTMLElement + Undef -> Bool ,
      ":disabled": HTMLElement + Undef -> Bool ,
      ":enabled": HTMLElement + Undef -> Bool ,
      ":even": HTMLElement + Undef -> Bool ,
      ":focus": HTMLElement + Undef -> Bool ,
      ":hidden": HTMLElement + Undef -> Bool ,
      ":odd": HTMLElement + Undef -> Bool ,
      ":tag": HTMLElement + Undef -> Str ,
      ":text": HTMLElement + Undef -> Bool ,
      ":trim": HTMLElement + Undef -> Bool ,
      ":unchecked": HTMLElement + Undef -> Bool ,
      ":visible": HTMLElement + Undef -> Bool ,
      #proto: Object, *: _, #code: _}
```

Makeable is ADsafe's whitelist of safe DOM elements.

```
type Makeable =
                 {a         : ${"a"},
                  abbr      : ${"abbr"},
                  acronym   : ${"acronym"},
                  address   : ${"address"},
                  area      : ${"area"},
                  b         : ${"b"},
                  bdo       : ${"bdo"},
                  big       : ${"big"},
                  blockquote: ${"blockquote"},
                  br        : ${"br"},
                  button    : ${"button"},
                  canvas    : ${"canvas"},
                  caption   : ${"caption"},
                  center    : ${"center"},
                  cite      : ${"cite"},
                  code      : ${"code"},
                  col       : ${"col"},
                  colgroup  : ${"colgroup"},
                  dd        : ${"dd"},
                  del       : ${"del"},
                  dfn       : ${"dfn"},
                  dir       : ${"dir"},
                  div       : ${"div"},
                  dl        : ${"dl"},
                  dt        : ${"dt"},
                  em        : ${"em"},
                  fieldset  : ${"fieldset"},
                  font      : ${"font"},
                  form      : ${"form"},
                  h1        : ${"h1"},
                  h2        : ${"h2"},
                  h3        : ${"h3"},
                  h4        : ${"h4"},
                  h5        : ${"h5"},
                  h6        : ${"h6"},
                  hr        : ${"hr"},
                  i         : ${"i"},
                  img       : ${"img"},
                  input     : ${"input"},
                  ins       : ${"ins"},
                  kbd       : ${"kbd"},
                  label     : ${"label"},
                  legend    : ${"legend"},
                  li        : ${"li"},
                  map       : ${"map"},
                  menu      : ${"menu"},
                  object    : ${"object"},
                  ol        : ${"ol"},
                  optgroup  : ${"optgroup"},
                  option    : ${"option"},
                  p         : ${"p"},
                  pre       : ${"pre"},
                  q         : ${"q"},
                  samp      : ${"samp"},
                  select    : ${"select"},
                  small     : ${"small"},
                  span      : ${"span"},
                  strong    : ${"strong"},
                  sub       : ${"sub"},
                  sup       : ${"sup"},
                  table     : ${"table"},
                  tbody     : ${"tbody"},
                  td        : ${"td"},
                  textarea  : ${"textarea"},
                  tfoot     : ${"tfoot"},
                  th        : ${"th"},
                  thead     : ${"thead"},
                  tr        : ${"tr"},
                  tt        : ${"tt"},
                  u         : ${"u"},
                  ul        : ${"ul"},
                  "var"     : ${"var"},
                  #proto    : Object,
                  *         : Bot,
                  #code     : Bot
              }
```

Appendix F

Desugared Lookup Function

This appendix shows the result of desugaring the following snippet of JavaScript to λ_{JS}:

```
function(obj, field) {
  if (field === "XMLHttpRequest") {
    return undefined;
  }
  else {
    return obj[field];
  }
}
```

For brevity, we do not show the object that wraps the function, but just the function below. We use the concrete syntax of λ_{JS} that is parsed by PLT Redex.

```
(lambda (this arguments)
  (let
   ((obj (get-field
          (deref (deref arguments))
          "0")))
   (let
    ((field (get-field
             (deref (deref arguments))
             "1")))
    (let
     ()
     (label $return
      (begin
       (begin
        (if (=== field "XMLHttpRequest")
         (begin
          (break $return
           (get-field
            (deref $global)
            "undefined"))
          undefined)
         (if (=== (typeof field) "string")
          (begin
(break $return
 ;;; The expression below (desugar 'obj[field])
 (get-field
  ;;; The expression below is (deref (desugar 'obj)).  If 'obj is
  ;;; not an object, it is converted to an object.  This is perfectly safe,
  ;;; since we just need it to have type JS.
  (deref (let
          (($2
            obj))
          (if (=== (typeof $2) "undefined")
           (throw ($makeException "TypeError" ":toObject received undefined"))
           (if (=== $2 null)
            (throw ($makeException "TypeError" ":toObject received null"))
            (if (=== (typeof $2) "boolean")
             (alloc (object ("$proto" $Boolean.prototype)
                            ("$class" "Boolean")
                            ("$value" $2)))
             (if (=== (typeof $2) "number")
              (alloc (object ("$proto" $Number.prototype)
                             ("$class" "Number")
                             ("$value" $2)))
              (if (=== (typeof $2) "string")
               (alloc (object ("$proto" $String.prototype)
                              ("$class" "String")
                              ("$value" $2)
                              ("length" (str-length $2))))
               $2)))))))
  ;;; The expression below is (toString (desugar 'field)), where toString
  ;;; is the metafunction defined in ECMA262-3, Section 9.8.1.  Of course,
  ;;; here we have toString written in JS.  In ECMA262-3, toString uses
  ;;; other metafunctions. Here, they are all inlined.
  (let
   (($toStr field))
   (if (=== (typeof $toStr) "location")
    ;;; The true branch below is unreachable.
    (if (if (=== (typeof (deref $toStr)) "object")
         #f
         #t)
     (throw "Catastrophe - toStr given a ref of a not-obj")
     (prim->string
       (let
         (($x $toStr))
         (if (=== (typeof $x) "location")
          (if (if (=== (typeof (deref $x)) "object")
               #f
               #t)
           (throw "Catastrophe - toPrim given a ref of a not-obj")
           (let
            (($vOf (get-field
                    (deref $x)
                    "valueOf")))
            (if (if (=== (typeof $vOf) "location")
                 (let
                  (($isF (deref $vOf)))
                  (if (=== (typeof $isF) "object")
                   (=== (typeof (get-field
                                 $isF
                                 "$code")) "lambda")
                   #f))
                 #f)
             (let
              (($vRes
                 (let
                   (($3
        $vOf))
      ;;; Below, field.valueOf() is called.  It may
      ;;; produce an arbitrary result, so it has type JS
      ;;; (not type NoXHR)
      ((get-field
        (deref $3)
        "$code") $x (alloc (alloc (object ("length" 0.0)
                                          ("callee" $3)
                                          ("$class" "Object")
                                          ("$proto" $Object.prototype)
                                              ("$isArgs" #t))))))))
 (if (prim? $vRes)
  $vRes
  (let
   (($toStr (get-field
             (deref $x)
             "toString")))
   (if (if (=== (typeof $toStr) "location")
        (let
         (($isF (deref $toStr)))
         (if (=== (typeof $isF) "object")
          (=== (typeof (get-field
                        $isF
                        "$code")) "lambda")
          #f))
        #f)
    (let
     (($tRes
        (let
          (($4
            $toStr))
          ;;; Below, field.toString() is called.
          ;;; A function application has type JS, not type NoXHR.
          ((get-field
            (deref $4)
            "$code") $x (alloc (alloc (object ("length" 0.0)
                                              ("callee" $4)
                                              ("$class" "Object")
                                              ("$proto" $Object.prototype)
                                              ("$isArgs" #t))))))))
     (if (prim? $tRes)
      $tRes
      (throw ($makeException "TypeError" ":apply expects array"))))
    (throw ($makeException "TypeError" ":apply expects array"))))))
(let
 (($toStr (get-field
           (deref $x)
           "toString")))
 (if (if (=== (typeof $toStr) "location")
      (let
       (($isF (deref $toStr)))
       (if (=== (typeof $isF) "object")
        (=== (typeof (get-field
                      $isF
                      "$code")) "lambda")
        #f))
      #f)
  (let
   (($tRes
      (let
       (($5
         $toStr))
       ((get-field
         (deref $5)
```

```
                          "$code") $x (alloc (alloc (object ("length" 0.0)
                                                            ("callee" $5)
                                                            ("$class" "Object")
                                                            ("$proto" $Object.prototype)
                                                            ("$isArgs" #t))))))))
                    (if (prim? $tRes)
                     $tRes
                     (throw ($makeException "TypeError" ":apply expects array"))))
                   (throw ($makeException "TypeError" ":apply expects array")))))))
              (if (=== (typeof $x) "object")
               (throw "Catastrophe - toPrim given plain object")
               $x)))))
        ;;; The expression below is the false branch of the conditional.  The
        ;;; type-test is unnecessary; it's inserted naively by desugaring.
        (if (=== (typeof $toStr) "object")
         (throw "Catastrophe - toStr given plain object")
         ;;; Here, we know that $toStr (i.e. field) has type NoXHR.  The operator
         ;;; prim->string is the identity on strings, and doesn't convert booleans or numbers to the
         ;;; string "XMLHttpRequest"!
         (prim->string $toStr))))))
    undefined)
   (begin
    (break $return
     (get-field
      (deref $global)
      "undefined"))
    undefined)))
 undefined)
undefined))))))
```

Bibliography

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

[2] Devdatta Akhawe, Adam Barth, Peifung E. Lam, John C. Mitchell, and Dawn Song. Towards a formal foundation of Web security. In *IEEE Computer Security Foundations Symposium*, 2010.

[3] Jong-hoon David An, Avik Chaudhuri, and Jeffrey S. Foster. Static typing for Ruby on Rails. In *IEEE International Symposium on Automated Software Engineering*, 2009.

[4] Davide Ancona, Massimo Ancona, Antonio Cuni, and Nicholas D. Matsakis. RPython: a step towards reconciling dynamically and statically typed OO languages. In *ACM SIGPLAN Dynamic Languages Symposium*, 2007.

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

[6] James P. Anderson. Computer Security Technology Planning Study. Technical Report ESD-TR-73-51, Deputy for Command and Management Systems, HQ Electronic Systems Division (AFSC), L. G. Handscom Field, Bedford, Massachusetts 01730, October 1972.

[7] Ihab Awad, Tyler Close, Adrienne Felt, Collin Jackson, Ben Laurie, Felix Lee, Ka-Ping Lee, David-Sarah Hopwood, Jasvir Nagra, Eric Sachs, Mike Samuel, Mike Stay, and David Wagner. Caja external security review. Technical report, Google Inc., 2008. `http://google-caja.googlecode.com/files/Caja_External_Security_Review_v2.pdf`.

[8] Sruthi Bandhakavi, Samuel T. King, P. Madhusudan, and Marianne Winslett. VEX: Vetting browser extensions for security vulnerabilities. In *USENIX Security Symposium*, 2010.

[9] Gavin Bierman, Erik Meijer, and Mads Torgersen. Adding dynamic types to C#. In *European Conference on Object-Oriented Programming*, 2010.

[10] Aaron Bohannon and Benjamin C. Pierce. Featherweight Firefox: Formalizing the core of a Web browser. In *USENIX Conference on Web Application Development*, 2010.

[11] A.H. Borning. Classes versus prototypes in object-oriented languages. In *ACM Fall Joint Computer Conference*, 1986.

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

[13] Kim B. Bruce, Luca Cardelli, and Benjamin C. Pierce. Comparing object encodings. *Information and Computation*, 155(1–2), 1999.

[14] Justin Cappos, Armon Dadgar, Jeff Rasley, Justin Samuel, Ivan Beschastnikh, Cosmin Barsan, Arvind Krishnamurthy, and Thomas Anderson. Retaining sandbox containment despite bugs in privileged memory-safe code. In *ACM Conference on Computer and Communications Security*, 2010.

[15] Ravi Chugh, Jeffrey A. Meister, Ranjit Jhala, and Sorin Lerner. Staged information flow for JavaScript. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2009.

[16] Ravi Chugh, Patrick M. Rondon, and Ranjit Jhala. Nested refinements for dynamic languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[17] Karl Crary, Stephanie Weirich, and Greg Morrisett. Intentional polymorphism in type-erasure semantics. In *ACM SIGPLAN International Conference on Functional Programming*, 1998.

[18] Douglas Crockford. ADSafe. `www.adsafe.org`, 2011.

[19] Tom Van Cutsem and Mark S. Miller. Proxies: Design principles for robust object-oriented intercession APIs. In *ACM SIGPLAN Dynamic Languages Symposium*, 2010.

[20] Andreas Dewald, Thorsten Holz, and Felix C. Freiling. ADSandbox: Sanboxing JavaScript to fight malicious websites. In *Symposium On Applied Computing*, 2010.

[21] Mohan Dhawan and Vinod Ganapathy. Analyzing information flow in JavaScript-based browser extensions. In *Computer Security Applications Conference*, 2009.

[22] ECMAScript language specification, 1999.

[23] ECMAScript language specification, 2009.

[24] Úlfar Erlingsson. *The Inlined Reference Monitor Approach to Security Policy Enforcement*. PhD thesis, Cornell University, 2003.

[25] Facebook. FBJS, 2011. `http://developers.facebook.com/docs/fbjs/`.

[26] Matthias Felleisen, Robert Bruce Findler, and Matthew Flatt. *Semantics Engineering with PLT Redex*. MIT Press, 2009.

[27] Matthew Finifter, Joel Howard Willis Weinberger, and Adam Barth. Preventing capability leaks in secure javascript subsets. In *Network and Distributed System Security Symposium*, 2010.

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

[29] Cormac Flanagan. ValleyScript: It's like static typing. Technical report, University of California, Santa Cruz, 2007. `http://users.soe.ucsc.edu/~cormac/papers/valleyscript.pdf`.

[30] Cormac Flanagan and Matthias Felleisen. Componential set-based analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.

[31] Cormac Flanagan, Matthew Flatt, Shriram Krishnamurthi, Stephanie Weirich, and Matthias Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

[32] David Flanagan and Yukihiro Matsumoto. *The Ruby Programming Language*. O'Reilly Media, 2008.

[33] Michael Furr, Jong-hoon David An, Jeffrey S. Foster, and Michael Hicks. Static type inference for Ruby. In *ACM Symposium on Applied Computing*, 2009.

[34] Michael Furr, Jong-hoon David An An, Jeffrey S. Foster, and Michael Hicks. The Ruby Intermediate Language. In *ACM SIGPLAN Dynamic Languages Symposium*, 2009.

[35] Google JavaScript style guide. `http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml`.

[36] James Gosling, Bill Joy, Jr. Guy Lewis Steele, and Gilad Bracha. *The Java Language Specification*. Addison Wesley, 3 edition, 2005.

[37] Salvatore Guarnieri and Benjamin Livshits. GateKeeper: Mostly static enforcement of security and reliability policies for JavaScript code. In *USENIX Security Symposium*, 2009.

[38] Arjun Guha, Shriram Krishnamurthi, and Trevor Jim. Static analysis for Ajax intrusion detection. In *International World Wide Web Conference*, 2009.

[39] Arjun Guha, Joe Gibbs Politz, and Shriram Krishnamurthi. Fluid object types. Technical Report CS-11-04, Brown University, 10 2011.

[40] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming*, 2010.

[41] Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi. Typing local control and state using flow analysis. In *European Symposium on Programming*, 2011.

[42] Philip Heidegger, Annette Bieniusa, and Peter Thiemann. Access permission contracts for scripting languages. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2012.

[43] Phillip Heidegger and Peter 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.

[44] Nevin Heintze. Control-flow analysis and type systems. In *International Static Analysis Symposium*, 1995.

[45] Fritz Henglein and Jakob Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *ACM SIGPLAN International Conference on Functional Programming*, 1995.

[46] David Herman. ClassicJavaScript. `www.ccs.neu.edu/home/dherman/javascript/`, 2005.

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

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

[49] Haruo Hosoya, Jérôme Vouillon, and Benjamin C. Pierce. Regular expression types for XML. *ACM Transactions on Programming Languages and Systems*, 27, 2005.

[50] Collin Jackson and Helen J. Wang. Subspace: Secure cross-domain communication for Web mashups. In *International World Wide Web Conference*, 2007.

[51] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Type analysis for JavaScript. In *International Static Analysis Symposium*, 2009.

[52] Simon Holm Jensen, Anders Møller, and Peter Thiemann. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*, 2010.

[53] Trevor Jim, Nikhil Swamy, and Michael Hicks. BEEP: Browser-enforced embedded policies. In *International World Wide Web Conference*, 2007.

[54] Emre Kıcıman and Benjamin Livshits. AjaxScope: A platform for remotely monitoring the client-side behavior of web 2.0 applications. In *Symposium on Operating System Principles*, 2007.

[55] Gary A. Kildall. A unified approach to global program optimization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1973.

[56] Casey Klein and Robert B. Finder. Randomized testing in PLT Redex. In *ACM SIGPLAN Workshop on Scheme and Functional Programming*, 2009.

[57] Mike Ter Louw, Karthik Thotta Ganesh, and V. N. Venkatakrishnan. AdJail: Practical enforcement of confidentiality and integrity policies on web advertisements. In *USENIX Security Symposium*, 2010.

[58] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In *Asian Symposium on Programming Languages and Systems*, 2008.

[59] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Isolating JavaScript with filters, rewriting, and wrappers. In *European Symposium on Research in Computer Security*, 2009.

[60] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Run-time enforcement of secure JavaScript subsets. In *Web 2.0 Security and Privacy*, 2009.

[61] Sergio Maffeis, John C. Mitchell, and Ankur Taly. Object capabilities and isolation of untrusted Web applications. In *IEEE Symposium on Security and Privacy*, 2010.

[62] Jonas Magazinius, Phu H. Phung, and David Sands. Safe wrappers and sane policies for self protecting JavaScript. In *OWASP AppSec Research*, 2010.

[63] Phillipe Meunier, Robert B. Findler, and Matthias Felleisen. Modular set-based analysis from contracts. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.

[64] Leo Meyerovich and Benjamin Livshits. ConScript: Specifying and enforcing fine-grained security policies for JavaScript in the browser. In *IEEE Symposium on Security and Privacy*, 2010.

[65] Leonid Mikhajlov and Emil Sekerinski. A study of the fragile base class problem. In *European Conference on Object-Oriented Programming*, 1998.

[66] Mark S. Miller, Mike Samuel, Ben Laurie, Ihab Awad, and Mike 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`.

[67] James George Mitchell. *The Design and Construction of Flexible and Efficient Interactive Programming Systems*. PhD thesis, Carnegie-Mellon University, 1970.

[68] Flemming Nielson and Hanne Riis Nielson. Type and effect systems. In *Correct System Design*. Springer, 1999.

[69] Jens Palsberg and Patrick O'Keefe. A type system equivalent to flow analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.

[70] Phu H. Phung, David Sands, and Andrey Chudnov. Lightweight self-protecting JavaScript. In *ACM Symposium on Information, Computer and Communications Security*, 2009.

[71] Benjamin C. Pierce. *Types and Programming Languages*. MIT Press, 2002.

[72] Joe Gibbs Politz, Spiridon Aristides Eliopoulos, Arjun Guha, and Shriram Krishnamurthi. ADsafety: Type-based verification of JavaScript sandboxing. In *USENIX Security Symposium*, 2011.

[73] Charles Reis, John Dunagan, Helen J. Wang, Opher Dubrovsky, and Saher Esmeir. BrowserShield: Vulnerability-driven filtering of dynamic HTML. In *Symposium on Operating Systems Design and Implementation*, 2006.

[74] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of JavaScript programs. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 2010.

[75] Amr Sabry and Matthias Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3), 1993.

[76] Claudiu Saftoiu. JSTrace: Run-time type discovery for JavaScript. Master's thesis, Brown University, 2010.

[77] Fred B. Schneider, Greg Morrisett, and Robert Harper. A language-based approach to security. In Reinhard Wilhelm, editor, *Informatics: 10 Years Back, 10 Years Ahead*, volume 2000 of *Lecture Notes in Computer Science*. Springer-Verlag, 2001.

[78] Olin Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.

[79] Gideon Joachim Smeding. An executable operational semantics for Python. Master's thesis, Utrecht University, 2009.

[80] T. Stephen Strickland, Sam Tobin-Hochstadt, and Matthias Felleisen. Practical variable-arity polymorphism. In *European Symposium on Programming*, 2009.

[81] Naoshi Tabuchi, Eijiro Sumii, and Akinori Yonezawa. Regular expression types for strings in a text processing language. *Electronic Notes in Theoretical Computer Science*, 75, 2003.

[82] Ankur Taly, Úlfar Erlingsson, Mark S. Miller, John C. Mitchell, and Jasvir Nagra. Automated analysis of security-critical JavaScript APIs. In *IEEE Symposium on Security and Privacy*, 2011.

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

[84] Peter Thiemann. A type safe DOM API. In *International Workshop on Database Programming Languages*, 2005.

[85] Sam Tobin-Hochstadt and Matthias Felleisen. Interlanguage migration: From scripts to programs. In *ACM SIGPLAN Dynamic Languages Symposium*, 2006.

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

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

[88] David Ungar and Randall B. Smith. SELF: The power of simplicity. In *ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications*, 1987.

[89] Mitchell Wand. Type inference for objects with instance variables and inheritance. In *Theoretical Aspects of Object-Oriented Progaramming*. MIT Press, 1994.

[90] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1), 1994.

[91] Andrew K. Wright and Robert Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1), 1997.

[92] Tobias Wrigstad, Francesco Zappa Nardelli, Sylvain Lebresne, Johan Östlund, and Jan Vitek. Integration of typed and untyped code in Thorn. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2010.

[93] Dachuan Yu, Ajay Chander, Nayeem Islam, and Igor Serikov. Javascript instrumentation for browser security. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2007.

[94] Chuan Yue and Haining Wang. Characterizing insecure JavaScript practices on the Web. In *International World Wide Web Conference*, 2009.

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