Typing Local Control and State Using Flow Analysis

Arjun Guha, Claudiu Saftoiu, and Shriram Krishnamurthi

Brown University

Abstract. Programs written in scripting languages employ idioms that confound conventional type systems. In this paper, we highlight one important set of related idioms: the use of local control and state to reason informally about types. To address these idioms, we formalize run-time tags and their relationship to types, and use these to present a novel strategy to integrate typing 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.

1 Introduction

"Scripting" languages are widely used in software development. Their lack of static types is touted as a positive feature that enables rapid prototyping. As programs grow large and complex, programmers need tools to reason about their code. A retrofitted type system, which would provide additional static guarantees, would help programmers manage evolution. However, care must be taken so that common idioms aren't deemed untypable; otherwise, either many programs or the languages themselves would have to change.

In section 2, we present examples from the aforementioned scripting languages that make heavy use of control and state to reason about "types". In section 3, we introduce a core calculus that can express the essence of these examples and a simple type system for this core calculus; but this type system alone cannot type-check our examples. In section 5, we present a program analysis that can reason about the idioms in our examples. One of our contributions is to clarify the relationship between static types and runtime tags (section 4), which scripting languages often confuse; we exploit this relationship to integrate type-checking with flow analysis in a tractable manner. In section 6, we present a simple proof of soundness for the combination of typing and flow analysis.

We have built an experimental type checker for JavaScript that uses these ideas. The implementation, discussion, and elided proofs, are available from `http://www.cs.brown.edu/research/plt/dl/flowtyping/v1/`.

```
1  // adapted from the Prototype library, v. 1.6.1
2  function serialize(val) {
3    switch (typeof val) {
4      case "undefined":
5      case "function":
6        return false;
7      case "boolean":
8        return val ? "true" : "false";
9      case "number":
10       return "" + val;
11     case "string":
12       return val;
13   }
14
15   if (val === null) { return "null"; }
16
17   var fields = [ ];
18   for (var p in val) {
19     var v = serialize(val[p]);
20     if (typeof v === "string") {
21       fields.push(p + ": " + v);
22     }
23   }
24   return "{ " + fields.join(", ") + " }";
25 }
```

Fig. 1. Non-local control in JavaScript

2 Control and State in Scripting Languages

We consider examples from canonical scripting languages. Our first example is the JavaScript function in figure 1, which serializes arbitrary values to strings. Functions and the special value `undefined` cannot be serialized, so for these it returns `false`. Since `val` may be any value, in a typed dialect of JavaScript, `serialize` should have a type equivalent to $\top \to \mathsf{Str} \cup \mathsf{Bool}$.

Let us informally reason about the type-safety of `serialize`. On line 3, the function branches on the result of `typeof val`. In JavaScript, the `typeof` operator[1] returns a string representing the "runtime type" of its argument. Thereafter:

- For `case "undefined"`, control falls through to line 5.
- On line 5, for `case "function"`, the function returns `false`.
- On line 7, 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 7.

[1] To be precise, `typeof` does not return a (static) type but a (runtime) tag. This distinction becomes significant in section 4.

```
1 # From the Python 2.5.2 standard libraries
2 def insort_right(a, x, lo=0, hi=None):
3     if hi is None:
4         hi = len(a)
5     while lo < hi:
6         mid = (lo+hi)//2
7         if x < a[mid]: hi = mid
8         else: lo = mid+1
9     a.insert(lo, x)
```

Fig. 2. Heap-sensitive reasoning in Python

– On line 9, for `case "number"`, the function uses string concatenation to coerce the number `val` to a string. Thus, `val` is a number because none of the preceding cases fall through to line 9.
– On line 11, for `case "string"`, the function returns `val`. Thus, `val` is a string because none of the preceding cases fall through to here.

This `switch` is missing a case! If `typeof val === "object"`, none of the earlier cases 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`, because 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 and 21 also employ flow-directed reasoning, but are relatively trivial. Therefore, we can conclude that `serialize` is safe.

Heap-Sensitive Reasoning Let us consider a Python example. The function `insort_right` (figure 2) inserts the argument `x` into the sorted array `a`, preserving sortedness. The additional optional arguments, `lo` and `hi`, are expected to be integers that specify the portion of array `a` to be returned.

The intended defaults are `lo=0` and `hi=len(a)`. However, the values of other arguments are not in scope when these expressions are evaluated. Therefore, `hi=len(a)` would signal an unbound identifier error. Instead, the program uses the default `hi=None` (which better guards against premature use than would a numeric default like `0`). The test on line 3 and the side-effect on line 4 ensure that `hi` is an integer in the continuation of the `if`-statement (lines 5–9). 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 `insort_right` by following their convoluted control-flow and side-effects, instead of merely following their syntactic structure. A reader may argue that these

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

Fig. 3. Tag Checks and Related Checks

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 [17] and is thus considered bad practice (e.g., [6]).

Irrespective of these options, the code above reflects what programmers do in practice. Figure 3 offers a conservative estimate of the prevalence of type tests and related checks across a broad corpus of 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 test for the presence of a field, because this operation is syntactically indistinguishable from field lookup.

Perspective The examples above make heavy use of local control and state to reason informally about "types". 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. Our novel *flow typing* system therefore separates typing from the account of flows and state. We present flow typing for an explicitly typed core calculus. We view type inference as a programmer convenience [9] that we leave for future work.

3 Semantics and Types

To present formal type and flow analysis systems, we have to settle on a runtime semantics. The languages under consideration have a kernel of higher-order functions and state that is essentially the same. This kernel is almost sufficient for our presentation, but we need to pick control operators and primitive operators, which do vary between languages. For now we will pick operators based on JavaScript, and return to this issue in section 5.

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

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

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

Fig. 4. Syntax and Semantics of λ_S

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

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

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

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

Fig. 5. Subtyping in λ_S

Figure 4 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 JavaScript-inspired primitives. Type annotations (discussed below) are ignored during evaluation. Although λ_S has first-class references, note that JavaScript, Python, and Ruby do not. However, first-class references allow us to model mutable variables and stateful objects [8, Section 2.1].

In this paper, 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. **break** 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 (figure 1). Other expressions, such as **tagof** `x !== "string"`, are a simple extension of our theory.

Figure 4 also specifies the syntax of types, T. Types include untagged unions and a top type $\top$, which were motivated in section 2. 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 (figure 5) is conventional.

Our typing relation is also mostly conventional. We present select typing judgments in figure 6. Note that the typing environment binds identifiers and labels. By T-SetRef, we can write subtypes to locations.[2] Finally, like JavaScript, λ_S programs cannot **break** across function boundaries, so we statically disallow it by dropping labels when typing functions (T-Abs).

4 Relating Static Types and Runtime Tags

Consider the following JavaScript program:

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

[2] This is a simple restriction of source and sink types [20, Chapter 15.5].

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

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

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

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

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

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

Fig. 6. Typing λ_S (Essential Rules)

```
  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:[3]

```
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 `-` 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 statically safe. Our goal is to enable the static type checker to arrive at the same conclusion.

To support such reasoning, a retrofitted type system must relate static types and runtime tags. We show this in figure 7. *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

[3] In earlier work, we desugared JavaScript to λ_{JS} in this form [8].

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

$$
\begin{array}{ll}
\multicolumn{2}{l}{runtime : T \to R} \\
runtime(\mathsf{Str}) & = \{\texttt{"string"}\} \\
runtime(\mathsf{Bool}) & = \{\texttt{"boolean"}\} \\
runtime(\mathsf{Num}) & = \{\texttt{"number"}\} \\
runtime(\mathsf{Undef}) & = \{\texttt{"undefined"}\} \\
runtime(S \cup T) & = runtime(S) \cup runtime(T) \\
runtime(S \cdots \to T) & = \{\texttt{"function"}\} \\
runtime(\bot) & = \emptyset \\
runtime(\top) & = r \\
runtime(\mathsf{Ref}\ T) & = \{\texttt{"location"}\}
\end{array}
$$

$$
\begin{array}{ll}
\multicolumn{2}{l}{static : R \times T \to T} \\
static(R, \mathsf{Str}) & = \mathsf{Str}, \text{if } \texttt{"string"} \in R \\
static(R, \mathsf{Bool}) & = \mathsf{Bool}, \text{if } \texttt{"boolean"} \in R \\
static(R, \mathsf{Num}) & = \mathsf{Num}, \text{if } \texttt{"number"} \in R \\
static(R, \mathsf{Undef}) & = \mathsf{Undef}, \text{if } \texttt{"undefined"} \in R \\
static(R, S \cdots \to T) & = S \cdots \to T, \text{if } \texttt{"function"} \in R \\
static(R, S \cup T) & = static(R, S) \cup static(R, T) \\
static(R, S \cup T) & = static(R, S), \text{if } static(R, T) \text{ is undefined} \\
static(R, S \cup T) & = static(R, T), \text{if } static(R, S) \text{ is undefined} \\
static(R, \top) & = \top \\
static(R, \mathsf{Ref}\ T) & = \mathsf{Ref}\ \top
\end{array}
$$

Fig. 7. Relationship Between Types and Tags

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 $\mathsf{Str} \cup \mathsf{Num}$ and its tag set is $\{\texttt{"number"}\}$, then *static* produces the type Num. Note that *static* is partial: for example, $static(\{\texttt{"number"}\}, \mathsf{Str})$ 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 (figure 8), which narrows the type of e based on the tag set R. By judiciously inserting **tagcheck**s, we can make our example typable.[4] We thus offer **tagcheck** as an appropriate cast-like operator for scripting languages.

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 **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:

[4] Section 5 presents an efficient technique to insert **tagcheck**s automatically, so they are hidden from the programmer.

$$e = \cdots \mid \textbf{tagcheck}\ \ R\ \ e \mid \textbf{tagerr}$$
$$E = \cdots \mid \textbf{tagcheck}\ \ R\ \ E$$

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

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

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

Fig. 8. Typing and Evaluation of Checked Tags

Lemma 1 (Preservation) *If $\Sigma, \cdot \vdash e : T$, $\Sigma \vdash \sigma$, and $\sigma e \to \sigma' e'$, then there exists a Σ', such that:*

i. $\Sigma', \cdot \vdash \sigma' e' : T$, *and*
ii. $\Sigma \subseteq \Sigma'$.

However, programs can get stuck on **tagerr**s:

Lemma 2 (Progress) *If $\Sigma, \cdot \vdash e : T$ and $\Sigma \vdash \sigma$, then either:*

i. $e \in v$, *or*
ii. there exist σ' and e', such that $\sigma e \to \sigma' e'$, or
iii. $e = E\langle \textbf{tagerr}\rangle$, *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 **tagcheck**s and **tagerr**s at all. Our key idea is to admit **tagerr**s to keep the type system simple, and then discharge them by other means.

5 Automatically Inserting Safe **tagcheck**s

We need a way to automatically insert **tagcheck**s 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 2. 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 7). 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 6).

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, the comments illustrate the kind of information we need from flow analysis:

```
1  let f = ref func(y) : Num ∪ Str → Num {
2    return:Num { /* tagof(y) ∈ {"number","string"} */
3      let x = ref y in /* x = ref y, tagof(y) ∈ {"number","string"} */
4      if (tagof (deref x) === "string") { /* same as line 3 */
5        break return 0 /* x = ref y, tagof(y) ∈ {"string"} */
6      }
7      else {
8        break return ((deref x) - 1) /* x = ref y, tagof(y) ∈ {"number"} */
9    } } }
10 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 6 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 10 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 typecheck 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 = ref y, tagof(y) ∈ {"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 6 simply restarts on all function applications. Thus we cannot insert a useful **tagcheck** around the subsequent **deref** `x`, so the example is untypable:

	JavaScript	Python	Ruby
Loops	✓	✓	✓
Exceptions	✓	✓	✓
Generators		✓	✓
Labelled Statements	✓		
Switch fall-through	✓		
Continuations			✓

Fig. 9. Control Features of Scripting Languages

```
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 = ref y, tagof(y) ∈ {"number", "string"} */}
in (deref g)("test")
```

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

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 soundness lemma (lemma 4).

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

6 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 (figure 9). 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.

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

Fig. 10. Syntax of λ_S in CPS

6.1 CPS Transformation

Figure 10 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 [21]. 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:

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

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" 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 figure 4. 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 3 (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.

$\widehat{S} : \hat{l} \to R$ abstract store
$\widehat{\Gamma} : x \to \widehat{V}$ abstract environments

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

$$\frac{R_1 \subseteq R_2}{R_1 \sqsubseteq R_2} \qquad \mathsf{LocTagof}\ \hat{l} \sqsubseteq \{\texttt{"string"}\} \qquad \mathsf{LocType}\ \hat{l}\ R \sqsubseteq \{\texttt{"boolean"}\}$$

$$\mathsf{Deref}\ \hat{l}\ R \sqsubseteq R \qquad \mathsf{Ref}\ \hat{l} \sqsubseteq \{\texttt{"location"}\}$$

Fig. 11. Analysis Domains

```
1  let f = func(k, y):(Num → ⊥) × Num ∪ Str → ⊥ {
2    // By V-Restart, k = {"function"}, y = {"number", "string"}
3    let x = ref y in // By F-Alloc, x = Ref l̂; l̂ = {"number", "string"}
4    let t1' = deref x in // By F-Deref, t' = Deref x Ŝ(l̂)
5    let t1 = tagcheck {"number","string"} t1' in // By F-TagCheck, t1 = t1'
6    let t2 = typeof t1' in // By F-Typeof, t2 = LocTypeof l̂
7    let t3 = (t2 === "string") in // By F-TypeIs-Str, t3 = LocType l̂ {"string"}
8      if (t3) { // By F-If-Split applied to l̂
9        k(0) } // By F-App, with l̂ = {"number"}
10     else {
11       let t4' = deref x in // By F-Deref, t4' = Deref x Ŝ(l̂); l̂ = {"number"}
12       let t4 = tagcheck {"number"} t4' in // By F-TagCheck, t4 = t4'
13       let t5 = t4' - 1 in
14         k(t5) } }
15 in let f' = deref f
16 in f'(k_init,200)
```

Fig. 12. tagcheck Insertion

6.2 Modular Flow Analysis

Figure 11 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, figure 12 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. 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 (figure 13).

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

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

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

$$
\text{(V-Const)}\ \widehat{\Gamma} \triangleright c \rightsquigarrow \delta_1(, c) \qquad \text{(V-Id)}\ \widehat{\Gamma} \triangleright x \rightsquigarrow \widehat{\Gamma}(x)
$$

$$
\text{(V-Sub)}\ \frac{\widehat{\Gamma} \triangleright V \rightsquigarrow \widehat{V} \qquad \widehat{V} \sqsubseteq \widehat{V'}}{\widehat{\Gamma} \triangleright V \rightsquigarrow \widehat{V'}}
$$

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

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

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

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

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

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

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

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

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

Fig. 13. Acceptability of Flow Analysis (Essential Rules)

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

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

Fig. 14. Assignment and Aliasing

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 figure 14, 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 figure 14 tackles aliasing in **ref** expressions. Similar rules apply to other syntactic forms.

Monotone Framework Our algorithm for computing tagchecks is a simple monotone framework [15] directly derived from the rules in figure 13. The monotone framework computes the abstract store and environment at each labelled expression. We use this information to insert **tagcheck**s into our programs.

Consider each expression of the form:

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

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

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

Soundness In addition to figure 13 and figure 14, we require trivial rules for cases where our flow analysis cannot determine useful information. These additional rules admit all other expressions, except **tagerr**s and possibly-faulty **tagcheck**s. 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 13 and 14 for clarity; in the following lemmas, we introduce it.

Lemma 4 (Soundness) *If* $\hat{S}, \cdot \vDash SM$ *and* $SM \to S'M'$ *then either:*

i. $\hat{S}', \cdot \vDash S'M'$, *or*
ii. M *is a* β_v*-redex,* $\mathsf{func}(x \cdots) : T \cdots \to \bot \ \{ \ N \ \}(V \cdots)$, *where for some* V, $\delta_1(\mathsf{tagof}, V) \notin runtime(T)$.

6.3 Combining Typing and Flow Analysis

We can now prove a stronger progress result that eliminates **tagerr**s.

Theorem 1 (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* $\sigma e \to \sigma' e'$.

Proof: This follows from lemma 2, with the possibility of **tagerr**s eliminated by inspection of figure 13—flow analysis does not admit expressions with **tagerr**s. ■

Theorem 1 requires a corresponding, combined preservation theorem.

Theorem 2 (Combined Preservation) *If:*

i. $\Sigma; \cdot \vdash e : T$,
ii. $\Sigma \vdash \sigma$,
iii. $\widehat{S}; \cdot \vDash \mathcal{P}_k[\![\sigma e]\!]$, *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. $\hat{S}'; \cdot \vDash \mathcal{P}_k[\![\sigma' e']\!]$.

Proof: Conclusions (i.), (ii.), and (iii.) follow immediately from lemma 1. For conclusion (iv.), apply lemma 3 to hypothesis (iv.) to get a reduction sequence, $\mathcal{P}_k[\![\sigma e]\!] \twoheadrightarrow \mathcal{P}_k[\![\sigma' e']\!]$. Apply lemma 4 at each step, eliminating case (ii.) of the lemma as follows. By lemma 3, intermediate expressions are not β_v-redexes, so case (ii.) does not apply. Suppose e itself has an active β_v-redex:

$$e = E\langle \mathsf{func}(x \cdots) \ : \ U \cdots \to S \ \{ \ e_f \ \}(v \cdots)\rangle$$

Once transformed to CPS, e has the form

$$\mathsf{func}(k, x \cdots) \ : \ (S \to \bot) \times U \cdots \to \bot \ \{ \ M_p \ \}(V \cdots)$$

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

$$\Sigma; \Gamma \vdash \mathsf{func}(x \cdots) \ : \ U \cdots \to S \ \{ \ e_f \ \}(v \cdots) : S$$

For all v, $\Sigma; \Gamma \vdash v : U$ by inversion. Hence $\delta_1(\mathbf{tagof}, v) \in \mathit{runtime}(U)$. Since conversion to CPS does not change tags, $\delta_1(\mathbf{tagof}, v) = \delta_1(\mathbf{tagof}, V)$, case (ii.) of lemma 4 does not apply. ■

7 Related Work

Typed Scheme Typed Scheme [23, 24] 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 2 and 4), which combine soundly (Theorems 1 and 2), 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** (figure 15)). We believe these are similar to transfer functions for dataflow analyses. However, Typed Scheme is not organized in this manner.

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

Fig. 15. If-splitting in Typed Scheme [23]

Intensional Polymorphism Intensional polymorphism [2] 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 [23], while our work relies on the relationship between static types and runtime tags (section 4).

Other Retrofitted Type Systems Soft Scheme [25] 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 paper and found in many scripting languages.

Anderson et al. [1] tackle type inference for JavaScript. However, their language is extremely limited, and their type system cannot tackle the idioms discussed in this paper (section 2).

Heidegger and Thiemann's [10] *recency types* account for ad hoc object initialization patterns that are pervasive in JavaScript, but does not address the problems that this paper 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 [12] 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" [12, Section 6.5], which is the goal of our work.

Diamondback Ruby [5] 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 [22, Chapter 9]. However, whole-program analysis for functional and object-oriented languages is non-modular and expensive [4] or difficult to make effective [3]. Meunier et al. [16] 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. However, unlike contracts, which are assumed to be correct, typing ensures that type annotations are correct. 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. [13, 14] and MrSpidey [4] 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 6.2).

There are known equivalences between various type systems and control-flow analyses, e.g., Heintze [11], Nielson and Nielson [18], and Palsberg and O'Keefe [19]. 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 (figure 6). We use a separate flow analysis to account for control-sensitive and heap-sensitive reasoning (section 5). We independently prove typing and flow analysis sound, then show that they combine in a simple way (section 6.3).

Definite assignment analysis is a commonly used flow analysis that augments typing (e.g., see the Java Language Specification [7, 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.

Acknowledgements

We thank Nicholas Cameron, Matthias Felleisen, and the anonymous reviewers for their careful comments on earlier drafts. We also thank Gilad Bracha, Cormac Flanagan, Jasvir Nagra, Steven Reiss, Ankur Taly, and Jan Vitek for enlightening discussions. This work is partially supported by the NSF and by Google.

References

1. C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *European Conference on Object-Oriented Programming*, 2005.
2. K. Crary, S. Weirich, and G. Morrisett. Intentional polymorphism in type-erasure semantics. In *ACM SIGPLAN International Conference on Functional Programming*, 1998.
3. C. Flanagan and M. Felleisen. Componential set-based analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1997.
4. C. Flanagan, M. Flatt, S. Krishnamurthi, S. Weirich, and M. Felleisen. Catching bugs in the web of program invariants. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, 1996.

5. M. Furr, J. D. An, J. S. Foster, and M. Hicks. Static type inference for Ruby. In *ACM Symposium on Applied Computing*, 2009.
6. Google JavaScript style guide. `http://google-styleguide.googlecode.com/svn/trunk/javascriptguide.xml`.
7. J. Gosling, B. Joy, J. G. L. Steele, and G. Bracha. *The Java Language Specification*. Addison Wesley, 3 edition, 2005.
8. A. Guha, C. Saftoiu, and S. Krishnamurthi. The essence of JavaScript. In *European Conference on Object-Oriented Programming*, 2010.
9. R. Harper and J. C. Mitchell. On the type structure of Standard ML. *ACM Transactions on Programming Languages and Systems*, 15(2), 1993.
10. P. Heidegger and P. Thiemann. Recency types for dynamically-typed, object-based languages: Strong updates for JavaScript. In *ACM SIGPLAN International Workshop on Foundations of Object-Oriented Languages*, 2009.
11. N. Heintze. Control-flow analysis and type systems. In *International Static Analysis Symposium*, 1995.
12. F. Henglein and J. Rehof. Safe polymorphic type inference for a dynamically typed language: Translating Scheme to ML. In *ACM SIGPLAN International Conference on Functional Programming*, 1995.
13. S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *International Static Analysis Symposium*, 2009.
14. S. H. Jensen, A. Møller, and P. Thiemann. Interprocedural analysis with lazy propagation. In *International Static Analysis Symposium*, 2010.
15. G. A. Kildall. A unified approach to global program optimization. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1973.
16. P. Meunier, R. B. Findler, and M. Felleisen. Modular set-based analysis from contracts. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2006.
17. L. Mikhajlov and E. Sekerinski. A study of the fragile base class problem. In *European Conference on Object-Oriented Programming*, 1998.
18. F. Nielson and H. R. Nielson. Type and effect systems. In *Correct System Design*. Springer, 1999.
19. J. Palsberg and P. O'Keefe. A type system equivalent to flow analysis. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 1995.
20. B. C. Pierce. *Types and Programming Languages*. MIT Press, 2002.
21. A. Sabry and M. Felleisen. Reasoning about programs in continuation-passing style. *LISP and Symbolic Computation*, 6(3), 1993.
22. O. Shivers. *Control-Flow Analysis of Higher-Order Languages*. PhD thesis, Carnegie Mellon University, 1991.
23. S. Tobin-Hochstadt and M. Felleisen. The design and implementation of Typed Scheme. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2008.
24. S. Tobin-Hochstadt and M. Felleisen. Logical types for untyped languages. In *ACM SIGPLAN International Conference on Functional Programming*, 2010.
25. A. K. Wright and R. Cartwright. A practical soft type system for Scheme. *ACM Transactions on Programming Languages and Systems*, 19(1), 1997.