Recency Types for Analyzing Scripting Languages

Phillip Heidegger and Peter Thiemann

Albert-Ludwigs-Universität Freiburg, Germany
{heidegger,thiemann}@informatik.uni-freiburg.de

Abstract. With the current surge of scripting technologies, large programs are being built with dynamically typed languages. As these programs grow in size, semantics-based tools gain importance for detecting programming errors as well as for program understanding.
As a basis for such tools, we propose a descriptive type system for an imperative call-by-value lambda calculus with objects. The calculus models essential features of JavaScript, a widely used dynamically-typed language: first-class functions, objects as property maps, and prototypes.
Our type system infers precise singleton object types for recently allocated objects. These object types are handled flow-sensitively and change during the objects' initialization phase. The notion of recency provides an automatic criterion to subsume these precise object types to summary object types, which are handled flow-insensitively. The criterion applies on a per-object basis. Thus, the type system identifies a generalized initialization phase for each object during which the change of its value is precisely reflected in the change of its type. Unlike with linear types, summary types may refer to singleton types and vice versa.
We prove the soundness of the type system and present a constraint-based inference algorithm. An implementation is available on the web.

1 Introduction

Dynamic features like animations, pop-down menus, and drag-and-drop are abundant in modern web applications. Many of these applications are implemented in JavaScript and rely on existing toolkits for realizing the dynamic features (Dojo, Prototype, Yahoo! UI, Scriptaculous, ...). However, these applications quickly become unmanageable because JavaScript provides neither namespace management nor data encapsulation. In addition, there are dubious language features which make it hard to write reliable programs [7].

Despite ongoing efforts in the construction of IDEs and debuggers, the development and maintenance of a JavaScript application suffers from numerous problems. Besides the aforementioned encapsulation issues, another host of problems is introduced by incompatible implementations of the browser's DOM-API by different vendors. A third major source of problems is the weak, dynamic type discipline of JavaScript and its prototype-based nature.

This paper concentrates on the typing aspect as type information can be helpful in all three problem areas. A type clash may hint at an encapsulation

```
1 var jstgp = { title: "JavaScript: The Good Parts" };
2 var dc = { name: "Douglas Crockford" };
3 // jstgp.author === undefined
4 // dc.book === undefined
5 jstgp.author = dc;
6 dc.book = jstgp;
7 jstgp.author.name;
```

(a) Bookstore.

```
1 function f (i, l) {
2   if (i == 0) return l;
3   var cell = new emptyCell();
4   cell.value = i;
5   cell.next = l;
6   return f (i−1, cell);
7 }
```

(b) List of integers.

Fig. 1. JavaScript examples.

problem; type information aids debugging and can provide hints to IDEs; and type information can contribute to addressing the incompatible browser problem. Last but not least, types are also useful for optimizing an implementation. As many other scripting languages (Python, Ruby, PHP, Lua, etc) are also dynamically typed and object-based, any progress in analyzing JavaScript programs can find fruitful application to analyzing programs in those languages as well.

1.1 Descriptive Types for Scripting Languages

To illustrate the problems that a type system for a scripting language must overcome, consider the JavaScript code in Fig. 1(a). The first two lines create objects with string-valued properties `title` and `name`. Lines 5 and 6 extend these objects with new properties that make the objects refer to one another. The last line accesses the `name` property of the `author` of the `jstgp` object.

Typing this code with a flow-insensitive type system would not be satisfactory, as each object would be assigned one type for its entire lifetime. Here is why: Accessing jstgp.author in line 3 yields undefined whereas the access in line 7 yields the dc object. Hence, the type of jstgp.author must be the top type $\top$ as it has to subsume both undefined and an object type. However, with this type, the expression jstgp.author.name is no longer type correct because the .name selector expects an object type as its argument.

Hence, *the type system must be flow-sensitive and admit strong updates* which change the type of an object according to the assignments to its properties. Strong update is vital to obtain precise object types in the presence of object extension and property update. Our type system achieves it by relying on *singleton object types*, which also keep track of aliasing to some extent. In line 3 of the example, the types of the objects are jstgp : obj(@1) and dc : obj(@2) where 1 and 2 are the abstract locations of these objects. A separate *heap type* specifies the contents of each object using a record type: $[@1 \mapsto \{\mathsf{title} : \mathsf{string}\}, @2 \mapsto \{\mathsf{name} : \mathsf{string}\}]$. While the variable types remain fixed in this example, the heap type changes at each line to $[@1 \mapsto \{\mathsf{title} : \mathsf{string}, \mathsf{author} : \mathtt{obj}(@2)\}, @2 \mapsto \{\mathsf{name} : \mathsf{string}, \mathsf{book} : \mathtt{obj}(@1)]$ at line 7. Thus, each assignment in the example *changes* the type of an object in the heap type and the type of jstgp.author.name in line 7 is string. This strong update is possible because each entry in the heap type describes exactly one object at run time.

In general, the type of an object cannot remain singleton and flow-sensitive throughout the whole program, as Fig. 1(b) shows. The function f builds a list-like structure where each cell is first created empty in line 3 and its properties are assigned subsequently. The type of cell starts out as obj(@3) and the heap type evolves from $[@3 \mapsto \{\}]$ to $[@3 \mapsto \{\mathsf{value} : \mathsf{number}, \mathsf{next} : t_l\}]$ after line 5, where t_l is the type of the argument l. As f invokes itself recursively, the type t_l must be related to the type of cell. If $t_l = \mathtt{obj}(@3)$, the object creation in line 3 would overwrite the information about l in the heap type, which is not correct.

To deal with such situations, our type system introduces *flow-insensitive summary object types*, which refer to a second heap type maintained by our system, the *summary heap type*. This heap type is globally available and it is fixed throughout the whole program. At any time, a singleton object type can be *demoted* into a summary object type at the price of losing strong updatability.

In the example, the type of the cell object must be demoted before passing the object as a parameter to f because the function allocates a new object at the same abstract location where the cell object resides. Hence, at line 6, demotion changes the type of cell to obj(˜3), the summary object type for location 3, removes the binding for @3 from the heap type, and requires the summary heap type to contain $[\tilde{}3 \mapsto \{\mathsf{value} : \mathsf{number}, \mathsf{next} : \mathtt{obj}(\tilde{}3)\}]$ (or at least to have ˜3 point to a suitable super type). With this demoted type of cell, there is now a consistent type for function f. Its argument types are number and obj(˜3) and its result type is obj(˜3), too. The corresponding summary heap type is $[\tilde{}3 \mapsto \{\mathsf{value} : \mathsf{number}, \mathsf{next} : \mathtt{obj}(\tilde{}3)\}]$. Thus, at any given point in a program, there may be up to two object types defined for each abstract location: the globally available summary heap type contains the type for an object of type $\mathtt{obj}(\tilde{}\ell)$ and the heap type *may* contain the information for $\mathtt{obj}(@\ell)$ if a suitable object exists.

1.2 Introducing Recency

The concept of recency [3] enables the automatic introduction of demotions. An object is recent after an execution step if it was either created in the execution step or if it was recent after the preceding step and the current step does not create a new object at the same abstract location.

Consequently, our type system assigns every new object a singleton object type. It keeps the type precise as long as it can guarantee that the object is recent. In Fig. 1(b), recency of the object at @3 gets lost at the recursive call of f, which may create a new object with abstract location 3.

Studies by Vitek and coworkers [18] as well as by Jensen and coworkers [15] have shown that recency fits the typical initialization pattern in JavaScript programs where the programmer allocates a number of objects and then initializes them. Quite often, no further properties are added after the initialization and the type of the properties rarely changes. During the initialization phase, each object is recent and its type can be updated in a flow-sensitive manner. Later on, when the object is no longer recent, its shape does not change anymore and it falls back to flow-insensitive typing.

1.3 Contributions and Outline

- We define a core calculus for scripting languages.
- We define a type system for the calculus that supports flow-sensitive and flow-insensitive typings for objects. Switching between the two modes is based on recency. Our system is the first formalization of recency with a type system in a higher-order setting.
- We prove type soundness. The proof is technically involved because recency-awareness requires a novel way of setting up the operational semantics including a non-standard substitution.
- We sketch the design of a type inference algorithm and provide an implementation on the web.[1]

§2 informally explains recency abstraction in terms of a type system and provides motivating examples. §3 defines syntax and dynamic semantics of the recency-aware core-calculus. §4 defines its static semantics. Next, §5 establishes the type soundness of the recency-aware type system. § 6 considers an extension to support prototypes, §7 gives an overview of our implementation, §8 discusses related work and §9 concludes.

A technical report with all proofs is available on the web [13].

2 Recency Typing, Informally

Recency is straightforward to handle in an abstraction interpretation setting [3, 15], but lifting the concept to types requires some care. There are four key points that need to be reflected in the design of the type system. First, there must be distinct types for recent objects and the remaining ones: singleton and summary object types. Second, singleton object types must be subject to strong update. Third, singleton object types must be "demotable" to summary types. Fourth, while an abstract interpreter can demote a singleton object to a summary object "online" at the respective `new` expressions, a type system must demote more conservatively to stay tractable.

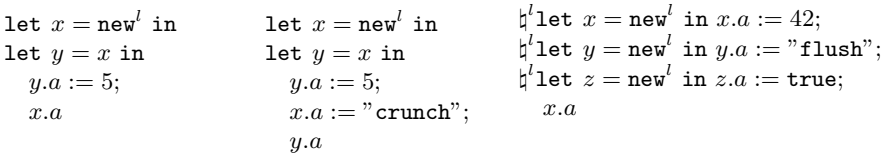
This section anticipates the syntax defined in §3, but writes $e_1; e_2$ for $\texttt{let}\ z = e_1\ \texttt{in}\ e_2$ ($z \notin \mathrm{fv}(e_2)$) and uses values of base types like `int`, `string`, and `bool` in the examples, although the formal calculus does not encompass them.

2.1 Types for Objects

There are two distinct kinds of object types, singleton ones $\texttt{obj}(@\ell)$ and summary ones $\texttt{obj}(\tilde{L})$, where the L stands for a set of locations. Locations are markers attached to each `new` expression in the program. Markers are usually distinct, but that is not required.

A first version of the typing judgment $\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} e : t \Rightarrow \Sigma', \Gamma'$ relates a summary heap environment Ω, a singleton heap environment Σ, a type environment Γ, and an expression with a type t and updated versions of the singleton

[1] `http://proglang.informatik.uni-freiburg.de/JavaScript/`.



(a) Aliasing. (b) Strong updates. (c) Summary Objects.

Fig. 2. Examples — objects.

environment Σ' and the typing environment Γ'. Both heap environments map abstract locations to object descriptions and the type environments map variables to types.

A singleton object type references an entry in the singleton environment. The example in Fig. 2(a) constructs an empty object, copies its reference to y, assigns to property a through y, and finally reads a through x. The typing for the final subterm $x.a$ of this expression is $\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} x.a : \mathtt{int} \Rightarrow \Sigma, \Gamma$ with $\Sigma = [l \mapsto [a \mapsto \mathtt{int}]]$ and $\Gamma = [x : \mathtt{obj}(@l), y : \mathtt{obj}(@l)]$. The typing environment Γ indicates that both, x and y, refer to the same object at l and the singleton environment Σ indicates that l refers to an object which has an a property of type `int` and which is otherwise undefined.[2]

Singleton object types enable strong update in the singleton environment as the example in Fig. 2(b) demonstrates. The typing for the final subterm $y.a$ is $\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} y.a : \mathtt{string} \Rightarrow \Sigma, \Gamma$ with $\Sigma = [l \mapsto [a \mapsto \mathtt{string}]]$ and $\Gamma = [x : \mathtt{obj}(@l), y : \mathtt{obj}(@l)]$. The assignment updates Σ, but it does not affect the object types. The typing of a property access dereferences the environment and yields the updated type.

Summary object types arise as soon as multiple objects are created with the same abstract location (Fig. 2(c)). Here is the typing for the final expression $x.a$:

$$\begin{gathered}\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} x.a : \top \Rightarrow \Sigma, \Gamma \\ \Omega = [l \mapsto [a \mapsto \top]], \Sigma = [l \mapsto [a \mapsto \mathtt{bool}]] \\ \Gamma = [x : \mathtt{obj}(\tilde{}\{l\}), y : \mathtt{obj}(\tilde{}\{l\}), z : \mathtt{obj}(@l)]\end{gathered}$$

The summary environment joins the types `int` and `string` for the property `a` to the top type $\top$. The summary object types contain the set $\{l\}$ in the example. There are singleton and summary object types for the same abstract location l.

The expression $\natural^L e$ is a *demotion* which changes the type of all L-objects from singleton to summary *before evaluating* e. In Fig. 2(c), a demotion appears at each $\mathtt{new}^l$ because an existing l-object loses its recency at that point. As our operational semantics maintains two distinct heaps (singleton and summary) corresponding to the distinct heap environments, a demotion actually moves objects from the singleton heap to the summary heap. The examples in this section contain explicit demotions to demonstrate their behavior, but a programmer never has to write demotions, because our analysis infers their placement.

[2] The summary environment Ω does not matter in this derivation.

```
♮^l let x = new^l in
let f = λ_.x.a in
x.a := 42;
♮^l let y = new^l in
y.a := 12;
♮^l let z = f(0) in z
```
(a) Functions.

```
♮^l let x = new^l in
let f = λ_.x.a in
x.a := 42;
♮^l(f(0))
```
(b) Effects 1.

```
♮^l let x = new^l in x.a := 42;
let g = λ_.new^l in
♮^l let y = g(0) in x.a
```
(c) Effects 2.

Fig. 3. Examples — functions.

2.2 Function Types

Let's turn to functions. Consider naively executing the example in Fig. 3(a). After the first line, x contains a reference to a newly allocated l-object in the singleton heap. The second line binds f to a closure, which captures this singleton reference and which is applied in line 6. However, before f is applied, the demotion in line 4 moves the l-object into the summary heap. This demotion is needed because a new l-object is created. However, the closure in f now contains a dangling reference to the singleton heap because the referenced object has been moved away. Hence the invocation of f in line 6 would result in a crash.

One approach to the problem would have demotion affect the captured references in a closure. However, changing a reference from singleton to summary affects its type and with it the type of the closure. So this approach is not viable.

The alternative, which we have opted for, is that free variables of functions never have singleton object types. We maintain this invariant by ensuring that substitution into a function body changes singleton references to summary references. This choice requires that the objects directly referenced in the free variables get demoted before the function is called. To ensure this demotion, the function type is equipped with an effect L that contains the locations of the object types in the free variables as well as the locations where the function allocates new objects. Each call site must perform a demotion $\natural^L$ before invoking the function. In Fig. 3(a), the type of f is $(\emptyset, \texttt{int}) \xrightarrow{\{l\}} (\emptyset, \texttt{int})$, a function that expects an integer and returns one, where the $\emptyset$ components state that no recent objects are passed to or returned from the function. Because the first l-object loses its recency with the demotion in line 4, the description of the l object in the summary heap is $\Omega(l) = [a \mapsto \texttt{int}]$. Hence, the return type of function f as well as the type of z is `int`.

The demotion $\natural^l$ in the last line is not required in this example, but it does hurt because it does not affect a reference captured in f, either. Our analysis inserts it because f has effect l. The example in Fig. 3(b) shows that the inserted demotion is needed, in general. Here, f is invoked without an intervening creation of a new l-object, so that the object captured by f is still recent at the function call. As the substitution of x into f has changed the captured singleton reference to a summary reference, the demotion in line 4 is needed to move the object into

the summary heap. Thus, when f is invoked its captured reference points to the correct heap.

In general, the type of a function is $(\Sigma_2, t_2) \xrightarrow{L} (\Sigma_1, t_1)$ where Σ_2 and Σ_1 are heap types for recent objects passed to the function and returned from it. The singleton heap type at the call site may contain more locations than Σ_2 and, similarly, Σ_1 specifies only the returned fragment of the heap type. L is the set of locations that the function body assumes demoted, as already discussed.

The last example, Fig. 3(c), shows the other use of effects. Function g allocates an object at the same location as the object pointed to by x, which is live and recent at the point where g is invoked. Inlining the function call exposes the $\texttt{new}^l$ expression. As this expression requires demotion to ensure correct execution of the expression $x.a$, the effect on the function arrow also contains all locations in which the expression allocates objects. Hence, the type of g is $(\emptyset, \texttt{int}) \xrightarrow{\{l\}} ([l \mapsto \{\ \}], \texttt{obj}(@l))$. The returned singleton environment describes the newly allocated l-object as everywhere undefined. The function returns a singleton reference to the newly allocated object.

While a singleton reference cannot be captured in a free variable of a function it can be passed as an argument, unless its location is in the function's effect.

$$\begin{array}{l} \natural^{l_1}\texttt{let } x = \texttt{new}^{l_1} \texttt{ in } x.a := 42; \\ \texttt{let } h = \lambda x.\, \big(\texttt{let } x_1 = \texttt{new}^{l_2} \texttt{ in } x_1.b := x;\ x_1\big) \texttt{ in} \\ \natural^{l_2}\texttt{let } y = h(x) \texttt{ in } x.a \end{array} \tag{1}$$

The type of h indicates that a singleton l_1-object is passed into the function body, an l_2-object is created, and an l_2-object which contains the l_1-object in property b is returned.

$$h : ([l_1 \mapsto [a : \texttt{int}]], \texttt{obj}(@l_1)) \xrightarrow{\{l_2\}} ([l_1 \mapsto [a : \texttt{int}], l_2 \mapsto [b : \texttt{obj}(@l_1)]], \texttt{obj}(@l_2))$$

3 The Recency-Aware Calculus

This section defines the syntax and semantics of the recency-aware calculus, $\mathcal{RAC}$. First some notation is needed. We write $\mathbf{Z}$ for the set of integers and $\mathrm{fv}(e)$ for free term variables in expression e. $A \xrightarrow{fin} B$ is the set of finite maps m from A to B, with $\{\}$ denoting the empty map. $\mathrm{dom}(m)$ is the domain of map m. $m \downarrow X$ restricts m to domain $\mathrm{dom}(m) \cap X$. $m \uparrow X$ restricts m to domain $\mathrm{dom}(m) \setminus X$. $m\{x \mapsto y\}$ is map update: if $m' = m\{x \mapsto y\}$, then $m'(x) = y$ and $m'(x') = m(x')$, if $x' \neq x$. $m\$a$ is property access for $m \in \mathsf{Prop} \xrightarrow{fin} \mathsf{Value}$. It is defined by $\{\}\$a = \texttt{udf}$, $m\{a \mapsto v\}\$a = v$, and $m\{b \mapsto v\}\$a = m\$a$, if $a \neq b$.

3.1 Syntax

Fig. 4 defines the syntax of expressions in A-normal form. A-normal form sequentializes the control flow inside a function. It slightly complicates the evaluation rule for `let`, but simplifies the typing rules and the proofs by reducing the number of context rules to the one for `let` expressions.

$$\begin{array}{rl} \mathsf{Value} \ni v & ::= x \mid \mathtt{rec}\, f(y).e \mid \mathtt{udf} \mid (q\ell, i) \\ \mathsf{TopExpr} \ni e & ::= v \mid \mathtt{let}^L\ x = s\ \mathtt{in}\ e \mid \natural^L e \\ \mathsf{Expr} \ni s & ::= v \mid v(v) \mid \mathtt{new}^\ell \mid v.a \mid v.a := v \end{array} \qquad \begin{array}{rl} \mathsf{Qualifier} \ni q & ::= \tilde{} \mid @ \\ \mathsf{Loc} \ni \ell & ::= l_1 \mid l_2 \mid \ldots \\ \mathsf{Loc} \supseteq L & \\ \mathsf{Prop} \ni a & \end{array}$$

Fig. 4. Expression syntax. Phrases marked in gray arise as intermediate steps during evaluation or are inserted automatically by elaboration.

A value v is either a variable, a recursive function abstraction, `udf` (undefined, a first order value), or a reference.[3] We write $\lambda y.e$ for $\mathtt{rec}\, f(y).e$ if $f \notin \mathrm{fv}(e)$. A reference value $(q\ell, i)$ refers to the object at address (ℓ, i) in the heap, where ℓ is the abstract location (allocation point) of the object and i is an integer unique among the objects created at ℓ. Source programs do not contain references, they arise only during execution.

A top-level expression e is either a value, a `let` expression, or a demotion. A `let` sequentializes the computation and its annotation L denotes the allocation effect of the header. Demotion $\natural^L e$ is discussed in §2.1. The effect annotation of the `let` expression guides demotion during substitution (see §3.2).

An expression s is either a value, a function call, an object creation, a property read, or a property write. The subexpressions of expressions are restricted to values. The expression $\mathtt{new}^\ell$ constructs a new object at location ℓ with no defined properties. The expressions $v.a$ for reading property a of object v and $v.a := v'$ for defining or updating property a of object v to be v' are standard.

3.2 Small-Step Operational Semantics

The semantics maintains objects in a heap, which is a mapping from heap addresses of the form $\mathsf{Loc} \times \mathbf{Z}$ to property maps (Fig. 5(a)). The heap consists of two parts with disjoint domains, the *singleton heap* for recent objects and the *summary heap* for the remaining objects. A singleton reference $(@\ell, i)$ refers to an object in the singleton heap, which contains at most one object for each location ℓ. The summary heap has no such restriction and its objects are referred to by summary references $(\tilde{}\ell, i)$.

A configuration of the semantics is a triple $K = (H, H_0, e)$ with H the summary heap, H_0 the singleton heap, and e an expression. Fig. 5(a) contains an inductive definition of the reduction relation $\longrightarrow$ on configurations.

The demotion rule SDem applies to $\natural^L e$ and moves all L-objects from the singleton heap to the summary heap. It relies on the demotion operation, which is defined for values, heaps, heap pairs, and property maps.

Demotion for values $v^{\natural L}$ (Fig. 5(b)) only affects singleton references with locations in L, which are converted to summary references. If L is omitted, then

[3] Variables are included in the syntactic category of values to obtain a concise definition of A-normal form syntax that is closed under reduction. All proofs assume closed values and thus rule out the variable case.

$$
\begin{array}{rcll}
H & \in & \mathsf{Heap} & = \mathsf{Loc} \times \mathbf{Z} \xrightarrow{fin} \mathsf{PropMap} \\
\mathcal{H} & \in & \mathsf{Heap} \times \mathsf{Heap} & \\
h & \in & \mathsf{PropMap} & = \mathsf{Prop} \xrightarrow{fin} \mathsf{Value} \\
\mathcal{L} & ::= & \Box \mid \mathtt{let}^L\ x = s\ \mathtt{in}\ \Box \mid \natural^L \Box &
\end{array}
$$

$$
\begin{array}{lll}
\textsc{SDem} & \mathcal{H}, \natural^L e & \longrightarrow \mathcal{H}^{\natural L}, e \\
\textsc{SApp} & \mathcal{H}, (\mathtt{rec}\, f(x).e)(v) & \longrightarrow \mathcal{H}, e\{f \Mapsto \mathtt{rec}\, f(x).e\}\{x \Mapsto v\} \\
\textsc{SLet} & \mathcal{H}, \mathtt{let}^L\ x = v\ \mathtt{in}\ e & \longrightarrow \mathcal{H}, e\{x \Mapsto v\} \\
\textsc{SNew} & H, H_0, \mathtt{new}^\ell & \longrightarrow H, H_0\{(\ell, i) \mapsto \{\}\}, (@\ell, i) \\
 & & \quad \text{if } \mathrm{dom}(H_0) \cap (\{\ell\} \times \mathbf{Z}) = \emptyset \\
 & & \quad \text{and } (\ell, i) \notin \mathrm{dom}(H) \\
\textsc{SRd} & \mathcal{H}, (q\ell, i).a & \longrightarrow \mathcal{H}, \mathcal{H}(q\ell, i)\$a \\
\textsc{SWrt} & \mathcal{H}, (q\ell, i).a := v & \longrightarrow \mathcal{H}\{(q\ell, i)(a) \mapsto v\}, \mathtt{udf}
\end{array}
$$

$$
\textsc{SLet}' \quad \frac{\mathcal{H}, s \longrightarrow \mathcal{H}', \mathcal{L}[v]}{\mathcal{H}, \mathtt{let}^L\ x = s\ \mathtt{in}\ e'' \longrightarrow \mathcal{H}', \mathcal{L}[\mathtt{let}^L\ x = v\ \mathtt{in}\ e'']}
$$

(a) Instrumented small-step operational semantics.

$$
\begin{array}{lll}
x^{\natural L} & = x & \\
(\mathtt{rec}\, f(x).e)^{\natural L} & = \mathtt{rec}\, f(x).e & (q\ell, i)^{\natural L} = \begin{cases} (\tilde{}\ell, i) & \text{if } \ell \in L \\ (q\ell, i) & \text{if } \ell \notin L \end{cases} \\
\mathtt{udf}^{\natural L} & = \mathtt{udf} &
\end{array}
$$

$$
\begin{array}{l}
(H, H_0)^{\natural L} = (H \cup H_L)^{\natural L}, (H_0 \backslash H_L)^{\natural L} \\
\qquad \text{where } H_L = H_0 \downarrow \{(l, i) \mid l \in L, i \in \mathbf{Z}\}
\end{array}
$$

(b) Demotion.

$$
(H, H_0)(q\ell, i) := \begin{cases} H(\ell, i) & \text{if } q = \tilde{} \\ H_0(\ell, i) & \text{if } q = @ \end{cases}
$$

$$
(H, H_0)\{(q\ell, i)(a) \mapsto v\} := \begin{cases} H\{(\ell, i)(a) \mapsto v\}, H_0 & \text{if } q = \tilde{} \\ H, H_0\{(\ell, i)(a) \mapsto v\} & \text{if } q = @ \end{cases}
$$

(c) Auxiliary definitions.

Fig. 5. Semantics.

$L = \mathsf{Loc}$. Demotion for heaps and property maps is defined pointwise. Demotion for a pair of heaps first moves all L-objects from the singleton heap H_0 to the summary heap and then applies heap demotion to both parts individually.

The rules SApp and SLet perform beta-value reduction, but with a non-standard notion of substitution defined in Fig. 6. As explained in §2.1, singleton references must not be substituted into the body of a lambda abstraction or a `let` expression with a conflicting allocation in its header (indicated by the L-annotation). The omitted cases just propagate the substitution to the subterms.

The rule SNew creates a new, empty ℓ-object in the singleton heap. The rule ensures that the address (ℓ, i) is unused in both heaps and that the singleton heap does not contain an ℓ-object, yet.

$$
\begin{array}{lcl}
(\natural^L e)\{x \Mapsto v\} & = & \natural^L(e\{x \Mapsto v^{\natural L}\}) \\
(\mathtt{rec}\, f(z).e)\{x \Mapsto v\} & = & \begin{cases} \mathtt{rec}\, f(z).e & \text{if } x \in \{z, f\} \\ \mathtt{rec}\, f(z).(e\{x \Mapsto v^{\natural}\}) & \text{if } x \notin \{z, f\} \end{cases} \\
(\mathtt{let}^L\ y = s\ \mathtt{in}\ e)\{x \Mapsto v\} & = & \begin{cases} \mathtt{let}^L\ y = s\{x \Mapsto v\}\ \mathtt{in}\ e & \text{if } x = y \\ \mathtt{let}^L\ y = s\{x \Mapsto v\}\ \mathtt{in}\ (e\{x \Mapsto v^{\natural L}\}) & \text{if } x \neq y \end{cases}
\end{array}
$$

Fig. 6. Substitution with demotion.

The rules SRD and SWRT read and write properties of objects. They make use of auxiliary read and write operations defined for pairs of heaps in Fig. 5(c).

The context rule SLET′ works because the result of reducing $s \in \mathsf{Expr}$ always has the form of some $e \in \mathsf{TopExpr}$, which can be written in the form $\mathcal{L}[v]$.

3.3 Properties of the Dynamic Semantics

The reduction relation maintains a number of invariants which hold for all configurations reachable from $(\emptyset, \emptyset, e)$ where e is a closed expression that does not contain object references. The proofs may be found in our technical report [13].

Lemma 1. *Let K be a configuration and $i \in \{1, 2, 3, 4, 5\}$.*
If $P_i(K)$ and $K \longrightarrow K'$, then $P_i(K')$.

1. *$P_1(H, H_0, e) \equiv fv(e) = \emptyset$. The expression e is closed.*
2. *$P_2(H, H_0, e) \equiv (\forall \ell) |dom(H_0) \cap (\{\ell\} \times \boldsymbol{Z})| \leq 1$. For each abstract location there exists at most one object in the singleton heap.*
3. *$P_3(H, H_0, e)$: For all expressions of the form $\mathtt{rec} f(x).e_0$ that occur in the configuration, the body e_0 does not contain a singleton reference $(@\ell, i)$.*
4. *$P_4(H, H_0, e)$: if $(@\ell, i)$ occurs in the configuration, then $(\ell, i) \in dom(H_0)$. A singleton reference refers to an object in the singleton heap.*
5. *$P_5(H, H_0, e) \equiv dom(H) \cap dom(H_0) = \emptyset$. The domains of the summary heap and the singleton heap are disjoint.*

4 Static Semantics

The type language defined in Fig. 7(a) distinguishes object types, the top type, the type `udf`, and function types. As explained in the informal part, an object type $\mathtt{obj}(p)$ refers to a record type r via the reference p, which either points to the summary heap $(\tilde{}L)$ or to the singleton heap $(@\ell)$. A record type r describes a finite map from properties $a \in \mathsf{Prop}$ to types. The productions for types t have a co-inductive reading to encompass recursive types without introducing an explicit μ-operator. They generate the set of regular trees where the leaves are marked with $\mathtt{obj}(p)$, $\top$, or `udf` and inner nodes with $\rightarrow$. Having recursive types does not imply that all proofs involving types require co-induction. Standard rule induction is sufficient except where otherwise noted.

$$
\begin{array}{lll}
t & ::= & \mathtt{obj}(p) \mid \top \mid \mathtt{udf} \mid (\Sigma, t) \xrightarrow{L} (\Sigma, t) \\
p & ::= & \tilde{}L \mid @\ell \qquad \text{with } |L| \geq 1 \\
r & ::= & \{\} \mid r\{a \mapsto t\} \\
\Omega & ::= & \emptyset \mid \Omega(\ell : r) \\
\Sigma & ::= & \emptyset \mid \Sigma(\ell : r) \\
\Gamma & ::= & \emptyset \mid \Gamma(x : t)
\end{array}
$$

(a) Type syntax.

$$
t <: t \qquad t <: \top \qquad \frac{L \subseteq L'}{\mathtt{obj}(\tilde{}L) <: \mathtt{obj}(\tilde{}L')}
$$

$$
\frac{t_1 <: t_1' \qquad t_2' <: t_2 \qquad L \subseteq L'}{(\Sigma_2, t_2) \xrightarrow{L} (\Sigma_1, t_1) <: (\Sigma_2, t_2') \xrightarrow{L'} (\Sigma_1, t_1')}
$$

(b) Subtyping.

Fig. 7. Type syntax and subtyping.

$\Omega, \Gamma \vdash_{\mathrm{v}} v : t$	value typing
$\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} e : t \Rightarrow L, \Sigma, \Gamma$	typing of expressions
$\Omega, \Sigma \vdash_{\mathrm{t}} t \lhd t$	flow from singleton to summary heap
$\Omega, \Sigma \vdash_{\mathrm{r}} @l.a : t$	read property
$\Omega, \Sigma \vdash_{\mathrm{w}} p.a := t \Rightarrow \Sigma'$	write property
$\Sigma, \Gamma \vdash_{c} L$	clean Σ and Γ from singleton L references

Fig. 8. Overview over relations

The function type $(\Sigma_2, t_2) \xrightarrow{L} (\Sigma_1, t_1)$ describes the calling context with singleton environment Σ_2 and the parameter with type t_2. The returned singleton environment Σ_1 is to replace Σ_2 at the call site and t_1 is the result type. The location set L on the arrow is the latent allocation effect of the function. On entry to the function, the singleton heap must not contain any L-object.

Fig. 8 lists the important judgments of the static semantics. The definitions are presented in Fig. 9-12 and explained in their respective section.

4.1 Subtyping

Fig. 7(b) defines the subtyping relation. Besides the rules dealing with reflexivity and top, one summary object type is a subtype of another, if it encompasses fewer locations. It is not possible to subsume a singleton object type to a summary object type. Function types are covariant in the return type and in the effects, but contravariant in the argument type. The singleton environments could be treated contra- and covariant analogously to the argument and return types, but are left invariant to keep the inference algorithm palatable.

Lemma 2. *The subtyping relation $<:$ is reflexive and transitive.*

Proof. Co-induction on the structure of types.

4.2 Typing of Values

Fig. 9 contains the typing rules for values. They derive the judgment $\Omega, \Gamma \vdash_{\mathrm{v}} v : t$ which relates the summary heap environment Ω, the type environment Γ, and a value with a type. The rules Undefined, Object, Variable, and

$$\textsc{Undefined}\quad \Omega, \Gamma \vdash_{\mathrm{v}} \mathtt{udf} : \mathtt{udf} \qquad \textsc{Object}\quad \Omega, \Gamma \vdash_{\mathrm{v}} (q\ell, i) : \mathtt{obj}(q\ell) \qquad \textsc{Variable}\quad \Omega, \Gamma \vdash_{\mathrm{v}} x : \Gamma(x)$$

$$\textsc{Subsumption}\ \frac{\Omega, \Gamma \vdash_{\mathrm{v}} v : t \quad t <: t'}{\Omega, \Gamma \vdash_{\mathrm{v}} v : t'} \qquad \textsc{Function}\ \frac{\mathrm{dom}(\Sigma) \cap L = \emptyset \quad L' \cup L'' \subseteq L \quad \Gamma' = \Gamma \downarrow \mathrm{fv}(\mathtt{rec}\, f(x).e) \quad L' = \mathrm{Locs}(\Gamma') \quad \Gamma'' = (\Gamma')^{\natural L'} \quad t_f = (\Sigma, t) \xrightarrow{L} (\Sigma', t') \quad \Omega, \Sigma, \Gamma''(f : t_f)(x : t) \vdash_{\mathrm{e}} e : t' \Rightarrow L'', \Sigma', \Gamma'''}{\Omega, \Gamma \vdash_{\mathrm{v}} \mathtt{rec}\, f(x).e : t_f}$$

Fig. 9. Typing rules for values.

Subsumption present no surprises. The rule Function is one of the main work horses of the type system. It has to ensure that a singleton reference for location ℓ does not sneak past allocations of new ℓ-references as illustrated in §2.2.

This problem has two facets, both of which are treated using effects [12]. First, a variable may hold a value with a singleton object type $\mathtt{obj}(@\ell)$ when a function is invoked which allocates a new object at ℓ (Fig. 3(c)). The solution is to equip each function type with an allocation effect, the set L'' of locations for which the function may allocate a new object. Through the condition $\mathrm{dom}(\Sigma) \cap L = \emptyset$, the function type insists that no L''-object is passed into the function as a singleton. (Recall that $L'' \subseteq L$.)

Second, our non-standard substitution demotes singleton references before transporting them into the body of a function. This demotion is conservative as shown in Fig. 3(b) and it causes most of the complication in the typing rule, but it seems that this complication is unavoidable.

► To see the problem addressed by non-standard substitution, consider an operational semantics which substitutes singleton references into functions. The example below shows the problems in trying to define a typing discipline for it:

$$\begin{array}{l} \natural^{\ell}\mathtt{let}\ x = \mathtt{new}^{\ell}\ \mathtt{in} \\ \mathtt{let}\ f = \lambda z.(x, z)\ \mathtt{in} \\ \natural^{\ell}\mathtt{let}\ y = \mathtt{new}^{\ell}\ \mathtt{in} \\ \natural^{\ell}\mathtt{let}\ w = f(y)\ \mathtt{in}\ w \end{array}$$

After line two, the type environment would contain $x : \mathtt{obj}(@\ell)$ and $f : ([\ell \mapsto \{\ \}], \mathtt{obj}(@\ell)) \xrightarrow{\emptyset} ([\ell \mapsto \{\ \}], \mathtt{obj}(@\ell) \times \mathtt{obj}(@\ell))$, which is consistent with the substitution of a singleton reference for x in the body of f. However, after line three these types have to change to $x : \mathtt{obj}(\tilde{}\ell)$ and $f : ([\ell \mapsto \{\ \}], \mathtt{obj}(@\ell)) \xrightarrow{\emptyset} ([\ell \mapsto \{\ \}], \mathtt{obj}(\tilde{}\ell) \times \mathtt{obj}(@\ell))$. While the change of x's type is a straightforward application of demotion, the change of f's type is not: it contains three occurrences of $\mathtt{obj}(@\ell)$, but only one of them must change to $\tilde{}\ell$. ◄

Integrating demotion into the typing rule is done with effects, again. Let Γ' be the typing environment of the function restricted to its free variables. The set $L' = \mathrm{Locs}(\Gamma')$ (see Fig. 10(a)) collects the locations mentioned in the types of the free variables of the function. It constructs the environment $\Gamma'' = (\Gamma')^{\natural L'}$ for the function body by demoting all types with respect to L', corresponding to

$$
\begin{aligned}
\mathrm{Locs}(\_ \xrightarrow{L} \_) &= \emptyset \\
\mathrm{Locs}(\top) &= \emptyset \\
\mathrm{Locs}(\texttt{udf}) &= \emptyset \\
\mathrm{Locs}(\texttt{obj}(\tilde{}L)) &= L \\
\mathrm{Locs}(\texttt{obj}(@l)) &= \{l\} \\
\mathrm{Locs}(\emptyset) &= \emptyset \\
\mathrm{Locs}(\Gamma(x:t)) &= \mathrm{Locs}(\Gamma) \cup \mathrm{Locs}(t)
\end{aligned}
$$

(a) Locations.

$$
\begin{aligned}
\texttt{obj}(@l)^{\natural L} &= \begin{cases} \texttt{obj}(\tilde{}\{l\}) & \text{if } l \in L \\ \texttt{obj}(@l) & \text{if } l \notin L \end{cases} \\
\texttt{obj}(\tilde{}L')^{\natural L} &= \texttt{obj}(\tilde{}L') \\
(\Sigma, t) \xrightarrow{L} (\Sigma', t')^{\natural L} &= (\Sigma, t) \xrightarrow{L} (\Sigma', t') \\
\top^{\natural L} = \top \qquad & \qquad \texttt{udf}^{\natural L} = \texttt{udf} \\
\emptyset^{\natural L} = \emptyset \qquad & \qquad \Gamma(x:t)^{\natural L} = \Gamma^{\natural L}(x : t^{\natural L})
\end{aligned}
$$

(b) Demotion of types and environments.

Fig. 10. Auxiliary functions.

the action of the substitution on references (Fig. 10(b)). It further adds L' to the effect of the function so that no L'-object is passed as a singleton to the function. Finally, the rule constructs the latent L-effect on the function arrow from the allocation effect L'' of the function body and L', the locations mentioned in the types of the free variables of the function (as discussed in §2.2).

4.3 Typing of Expressions

Fig. 11 contains the typing rules for expressions, which were informally explained in §2. They rely on some auxiliary judgments defined in Fig. 12. The VALUE rule expresses that the evaluation of a value has no effect. Hence, the environments pass through unchanged and the effect is empty.

The rule LET sequentializes computations. The effect L_1 of the header becomes the annotation of the `let`, which is inferred by our analysis. The L-effects of header and body are merged to obtain the effect of the whole expression. The type and singleton environments are threaded through the header and through the body. The condition $\Sigma, \Gamma \vdash_c L_1$ determines the set of locations that must **not** be passed precisely into the `let` header.

The FUNCTION CALL rule is driven by the L-effect of the function type. It requires that Γ and Σ do not contain singleton L-references (ensured by $\Sigma, \Gamma \vdash_c L$). It matches the expected argument type with the type of v_2 and yields the return type t_1.

The DEMOTE rule types the demotion $\natural^L$. It reflects the move of the L-objects from the singleton heap to the summary heap using the *flow relation* $\Omega, \Sigma \vdash_{\mathrm{t}} t \lhd t'$ (defined in Fig. 12), where t is the original type and t' is its demoted counterpart. Furthermore, the typing rule demotes references in Σ and Γ and ensures that Ω does not contain singleton L-references.[4]

The NEW rule types object creation. As objects are always created as singleton objects without predefined properties, the return type is $\texttt{obj}(@\ell)$ and the singleton heap environment registers the binding $\ell \mapsto \{\}$. The other preconditions guarantee that no singleton ℓ-reference exists when the new ℓ-object is created.

[4] Ω may contain singleton references for locations which are never demoted.

$$\textsc{Value}\quad \frac{\Omega, \Gamma \vdash_{\mathrm{v}} v : t}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} v : t \Rightarrow \emptyset, \Sigma, \Gamma} \qquad \textsc{New}\quad \frac{\Sigma, \Gamma \vdash_{c} \{\ell\} \qquad \ell \in \mathrm{dom}(\Omega)}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} \mathtt{new}^{\ell} : \mathtt{obj}(@\ell) \Rightarrow \{\ell\}, \Sigma(\ell : \{\}), \Gamma}$$

$$\textsc{Let}\quad \frac{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} s_1 : t_1 \Rightarrow L_1, \Sigma_1, \Gamma_1 \qquad \begin{array}{c}\Sigma, \Gamma \vdash_{c} L_1 \\ \Omega, \Sigma_1, \Gamma_1(x : t_1) \vdash_{\mathrm{e}} e_2 : t_2 \Rightarrow L_2, \Sigma_2, \Gamma_2(x : t_1')\end{array}}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} \mathtt{let}^{L_1}\ x = s_1\ \mathtt{in}\ e_2 : t_2 \Rightarrow L_1 \cup L_2, \Sigma_2, \Gamma_2}$$

$$\textsc{Function Call}\quad \frac{\begin{array}{c}\Sigma, \Gamma \vdash_{c} L \qquad \Omega, \Gamma \vdash_{\mathrm{v}} v_2 : t_2 \\ \Omega, \Gamma \vdash_{\mathrm{v}} v_1 : (\Sigma, t_2) \xrightarrow{L} (\Sigma', t_1)\end{array}}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} v_1(v_2) : t_1 \Rightarrow L, \Sigma', \Gamma} \qquad \textsc{Demote}\quad \frac{\begin{array}{c}\Omega, \Sigma', \Gamma^{\natural L} \vdash_{\mathrm{e}} e : t \Rightarrow L', \Sigma'', \Gamma'' \\ \Sigma' = \Sigma^{\natural L} \uparrow L \qquad L \subseteq L' \qquad \Omega = \Omega^{\natural L} \\ \forall l \in L : \Omega, \Sigma \vdash_{\mathrm{t}} \mathtt{obj}(@l) \lhd \mathtt{obj}(\tilde{}l)\end{array}}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} \natural^{L} e : t \Rightarrow L', \Sigma'', \Gamma''}$$

$$\textsc{Read}\quad \frac{\Omega, \Gamma \vdash_{\mathrm{v}} v : \mathtt{obj}(p) \qquad \Omega, \Sigma \vdash_{\mathrm{r}} p.a : t}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} v.a : t \Rightarrow \emptyset, \Sigma, \Gamma} \qquad \textsc{Write}\quad \frac{\Omega, \Gamma \vdash_{\mathrm{v}} v : \mathtt{obj}(p) \qquad \Omega, \Gamma \vdash_{\mathrm{v}} v' : t' \qquad \Omega, \Sigma \vdash_{\mathrm{w}} p.a := t' \Rightarrow \Sigma'}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} v.a := v' : \mathtt{udf} \Rightarrow \emptyset, \Sigma', \Gamma}$$

Fig. 11. Typing rules for expressions.

The Read rule relies on an auxiliary judgment (see Fig. 12) that performs a lookup either in the summary environment or in the singleton one, depending on the precision of the reference. The Write rule is similar. Its auxiliary judgment returns a new singleton environment because it performs a strong update on singleton object types. The return type is `udf`.

4.4 Auxiliary Judgments

Fig. 12 defines auxiliary judgments used in the typing rules. The first group of rules defines the flow judgment $\Omega, \Sigma \vdash_{\mathrm{t}} t \lhd t'$. The interesting part of the flow relation is how it relates a singleton object type to a summary object type. It does so by imposing a subtyping constraint between the property maps of the two types. Thus, flow takes a snapshot of (part of) the current state of the singleton environment and joins it into the summary heap environment.

Flow is invoked from the Demote rule to convert a reference from the singleton environment Σ to the summary environment Ω (from t to t'). The rule for the object type is the most important one: If an ℓ-reference changes from singleton to summary, then the corresponding entries in Σ and Ω must be flow-related. The rule for record types pushes the flow into the object's properties, and the remaining rules relate the property types by subtyping. For example, if $\Sigma(\ell)\$a = \mathtt{udf}$, then $\Omega(\ell)\$a$ can be either `udf` or $\top$.

The second group of rules defines the read judgment $\Omega, \Sigma \vdash_{\mathrm{r}} p.a : t$, which executes the abstract read operation of property a from reference p. The type of a singleton or summary reference with a single location comes straight from the

$$\frac{t <: t'}{\Omega, \Sigma \vdash_{\mathrm{t}} t \lhd t'} \qquad \frac{l \notin \mathrm{dom}(\Sigma)}{\Omega, \Sigma \vdash_{\mathrm{t}} \mathtt{obj}(@l) \lhd \mathtt{obj}(\tilde{}l)} \qquad \frac{(\forall a \in \mathsf{Prop}) \quad \Omega, \Sigma \vdash_{\mathrm{t}} \Sigma(l)(a) \lhd \Omega(l)(a)}{\Omega, \Sigma \vdash_{\mathrm{t}} \mathtt{obj}(@l) \lhd \mathtt{obj}(\tilde{}l)}$$

$$\frac{\Sigma(l)(a) = t}{\Omega, \Sigma \vdash_{\mathrm{r}} @l.a : t} \qquad \frac{(\forall l \in L)\ \Omega(l)(a) <: t}{\Omega, \Sigma \vdash_{\mathrm{r}} \tilde{}L.a : t} \qquad \frac{(\forall \ell \in L)\ t <: \Omega(\ell)(a)}{\Omega, \Sigma \vdash_{\mathrm{w}} \tilde{}L.a := t \Rightarrow \Sigma}$$

$$\frac{\Sigma' = \Sigma[l, a \mapsto t]}{\Omega, \Sigma \vdash_{\mathrm{w}} @l.a := t \Rightarrow \Sigma'} \qquad \frac{\Gamma = \Gamma^{\natural L} \qquad \Sigma = \Sigma^{\natural L} \qquad \mathrm{dom}(\Sigma) \cap L = \emptyset}{\Sigma, \Gamma \vdash_c L}$$

Fig. 12. Rules for flow, reading, and writing to the heap.

respective environment. The type of a reference spread over locations in L is a supertype of the types at each location.

The third group defines the write judgment $\Omega, \Sigma \vdash_{\mathrm{w}} p.a := t \Rightarrow \Sigma'$, which performs a write operation to property a of reference p. The value to be stored has type t, and the write returns a modified singleton environment Σ' in case p is a singleton reference. Writing to a summary reference (first rule) places a subtyping constraint: the type in the summary environment must subsume the type t of the written value. Writing to a singleton reference does not affect the summary environment, but changes the respective location in the singleton environment.

The last rule governs the interplay between allocation effects, the variable environment, and the singleton environment. It states that the singleton environment must not contain entries from the allocation effect L and that neither environment contains precise L-references.

Modeling the read and write operations in auxiliary judgments enables us to extend them modularly to support conditionals and prototypes (see our technical report [13]).

5 Metatheory

This section presents the type soundness result for $\mathcal{RAC}$. The basic structure of the proof follows Felleisen and Wright [26], where type soundness follows from a type preservation and a progress lemma. The progress lemma turns out to be surprisingly hard to establish. Beyond mere typing, it requires a number of invariants about program execution.

For preservation and progress, we extend the notion of typing to configurations with closed expressions. $\Omega, \Sigma \Vdash_e H, H_0, e : t \Rightarrow L, \Sigma'$ ensures the consistency between the dynamic heaps H, H_0 and the static heap abstraction Ω, Σ in addition to the type judgment $\Omega, \Sigma, \emptyset \vdash_{\mathrm{e}} e : t \Rightarrow L, \Sigma', \emptyset$. Consult the technical report [13] for the full definitions.

Lemma 3 (Substitution). *Suppose that* $\Omega, \emptyset \vdash_{\mathrm{v}} v : t_x$ *and* $\Omega, \Sigma, \Gamma(x : t_x) \vdash_{\mathrm{e}} e : t_0 \Rightarrow L, \Sigma', \Gamma'$.

Then $\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} e\{x \Rightarrow v\} : t_0 \Rightarrow L, \Sigma', \Gamma'$.

Lemma 4 (Preservation). *Suppose that* $H, H_0, e \longrightarrow H', H_0', e'$ *and* $\Omega, \Sigma \Vdash_e H, H_0, e : t \Rightarrow L, \Sigma'$. *Then there exists some* Σ_n *and* $L_n \subseteq L$ *with*

$$\Omega, \Sigma_n \Vdash_e H', H_0', e' : t \Rightarrow L_n \Sigma'.$$

Lemma 5 (Progress). *Suppose that* $\Omega, \Sigma \Vdash_e H, H_0, e : t \Rightarrow L, \Sigma'$ *and there exists an expression* e_0, *such that* $\emptyset, \emptyset, e_0 \longrightarrow^* H, H_0, e$. *Then either* e *is a value or there exists* H', H_0', e' *such that* $H, H_0, e \longrightarrow H', H_0', e'$.

6 Prototype Extension

It is fairly straightforward to extend $\mathcal{RAC}$ with a JavaScript-style prototype mechanism. Fig. 13 defines the calculus $\mathcal{RAC}'$ as an extension to syntax, operational semantics, and typing of $\mathcal{RAC}$. $\mathcal{RAC}'$ has an enhanced version of object creation, $\texttt{new}^{\ell}(v)$. Its reduction rule SNEW$'$ initializes a reserved prototype property `_p` of the new object to v. This property must not be used in user code.

The read reduction rule SRD$'$ replaces the SRD rule in Fig. 5. The read operation first examines the value v obtained by reading the property directly from the object itself. It returns v if $v \neq \texttt{udf}$. Otherwise, if a prototype is defined for the object, it delegates the lookup to the prototype. If the property is undefined or the prototype is not an object, the read operation returns `udf`.

The revised typing rule for `new` installs the prototype argument in the newly created object. The prototype argument is an arbitrary value.

All other typing rules remain the same, but the auxiliary judgment to read a property needs to be revised. It mimics the operational semantics in descending the prototype chain of the object, returning the value when the property is found, and recursively reading the prototype if one exists.

Writing of a property is not affected by prototypes because the write operation only affects the top-level object and ignores the prototype chain [8].

7 Implementation

Type inference for the type system is decidable. A non-deterministic algorithm guesses for each demotion the set of abstract locations. This is possible for each program, because it only contains a finite set of abstract locations. After this guess typing becomes straightforward, because the demotions determine at each program point if the program accesses the singleton heap or the summary heap. As this algorithm is forbiddingly expensive, we designed a constraint-based implementation.

Our prototype implementation consists of roughly 9000 lines of OCaml code. It uses a syntax-directed version of the type system to generate constraints (essentially equality, subtyping, flow, and map constraints, where the latter constrain the domain of a singleton environment). The solver for these constraints

Additional syntax

$$e ::= \cdots \mid \texttt{new}^{\ell}(v)$$

Additional reductions

$$\begin{array}{lll}
\textsc{SNew}' & H, H_0, \texttt{new}^{\ell}(v) \longrightarrow H, H_0[(\ell, j) \mapsto \{_\texttt{p} \mapsto v\}], (@\ell, j) & \\
& \quad \text{if } \mathrm{dom}(H_0) \cap \{\ell\} \times \mathbf{Z} = \emptyset & \\
& \quad \text{and } (\ell, j) \notin \mathrm{dom}(H) & \\
\textsc{SRd}' & H, H_0, (p, i).a \longrightarrow H, H_0, \mathit{read}(H, H_0, (p, i), a) &
\end{array}$$

$$\mathit{read}(H, H_0, (p, i), a) = \begin{cases} v & \text{if } v \neq \texttt{udf} \\ (q, i).a & \text{if } v = \texttt{udf} \wedge pt = (q, i) \\ \texttt{udf} & \text{otherwise} \end{cases}$$

$$\begin{array}{lll}
\text{where} & pt = (H, H_0)(p, i)\$_\texttt{p} & \\
& v = (H, H_0)(p, i)\$a &
\end{array}$$

Changes in the static semantics

$$\textsc{New'} \quad \frac{\Sigma, \Gamma \vdash_c \ell \qquad \ell \in \mathrm{dom}(\Omega) \qquad \Omega, \Sigma, \Gamma \vdash_{\mathrm{v}} v : t}{\Omega, \Sigma, \Gamma \vdash_{\mathrm{e}} \texttt{new}^{\ell}(v) : \texttt{obj}(@\ell) \Rightarrow \{\ell\}, \Sigma(\ell \mapsto \{\_\texttt{p} \mapsto t\}), \Gamma} \qquad \frac{t <: t' \qquad (\forall \ell \in L)\ \Omega, \Sigma \vdash_{\mathrm{r}} \tilde{}\ell.a : t}{\Omega, \Sigma \vdash_{\mathrm{r}} \tilde{}L.a : t'}$$

$$\frac{(\Omega, \Sigma)(p)(a) = t \neq \texttt{udf}}{\Omega, \Sigma \vdash_{\mathrm{r}} p.a : t} \qquad \frac{(\Omega, \Sigma)(p)(a) = \texttt{udf} \qquad (\Omega, \Sigma)(p)(\_\texttt{p}) = \texttt{obj}(q) \qquad \Omega, \Sigma \vdash_{\mathrm{r}} q.a : t}{\Omega, \Sigma \vdash_{\mathrm{r}} p.a : t}$$

$$\frac{(\Omega, \Sigma)(p)(a) = \texttt{udf} \qquad (\Omega, \Sigma)(p)(\_\texttt{p}) \neq \texttt{obj}(p')}{\Omega, \Sigma \vdash_{\mathrm{r}} p.a : \texttt{udf}}$$

Fig. 13. Extension to support prototypes.

builds a hyper-graph where variables (e.g., type variables α, location set variables μ) are vertices and constraints are hyperedges.

The implementation of the inference algorithm is available on the web at `http://proglang.informatik.uni-freiburg.de/JavaScript/`. The implementation infers the types and locations for all examples in the paper.

Figure 14 introduces variables and constraints. Type variables (α) are mapped to types by type inference. During inference we map them to two types (indicated by $[\mapsto t, t]$). The first type is a lower bound, the second one is an upper bound (with respect to the subtyping relation). Location variables (μ) and precision variables (χ) are mapped to suitable values during constraint simplification. Object variables (ω) are assigned to maps from properties to type variables. The singleton environment variables (σ) are assigned to maps from abstract locations to object type variables. The need of the upper and lower bound for variables is explained in the section on constraint simplification 7.3.

The semantics of the constraints is given with respect to the relations defined for the logical type system. See the technical report [13] for details.

$$
\begin{array}{lll}
\alpha & [\mapsto t, t &] \text{ type variable} \\
\mu & [\mapsto L, L &] \text{ location variable} \\
\chi & [\mapsto q, q &] \text{ precision variable} \\
\omega & [\mapsto \mathsf{Prop} \xrightarrow{fin} \alpha &] \text{ object variable} \\
\sigma & [\mapsto \mathsf{Loc} \xrightarrow{fin} \omega &] \text{ local environment variable}
\end{array}
$$

$$
\begin{array}{llll}
\tau & ::= & t \mid \alpha & \text{extended types} \\
S & \in & \{\in, \notin\} & \text{basic set operations} \\
Y & \in & \{\tau, \mu, \chi, \omega, \sigma\} & \text{wild-card} \\
C & & & \text{constraints}
\end{array}
$$

$\tau <: \tau$	$\sigma \vdash \tau \lhd \tau$		subtype, flow
$l\ S\ \mu$	$\mu \subseteq \mu$	$\mu \cap \mu = \emptyset$	sets
$\sigma =^{\#}_{\mu} \sigma$	$\alpha =^{\#}_{\mu} \alpha$	$\omega =^{\#}_{\mu} \omega$	demotion
$\sigma \vdash_{\mathrm{r}} \alpha.a : \alpha$	$\sigma \vdash_{\mathrm{w}} \alpha.a := \alpha \Rightarrow \sigma$		property access
$\sigma = \sigma[\ell \mapsto \{\}]$	$\mu = \mathrm{Locs}(\tau)$		references
$Y = Y$	$C \wedge C$	False	

Fig. 14. Constraint Syntax

7.1 Preprocessing

A preprocessing step inserts demotions. It wraps a demotion $\natural^L$ around every `let` expression the right hand side of which is either a function call or a `new` expression. The L-annotation of the let is either a location variable μ (in case of the function call), or the singleton set $L = \{l\}$, where l is the annotation of the new expression. The content of the location variable μ is inferred during type inference. The preprocessing step is required. Type inference cannot complete successfully without it.

7.2 Constraint Generation

Figure 15 and Figure 16 contain the most important constraint generation rules. In the inference system the summary heap environment Ω is a map from abstract locations to type variables. The type environment Γ maps variables to type variables as usual. The domain of the type environment is known statically due to static scoping of our core language. Hence a constraint like $\bigwedge_{x \in \mathrm{dom}(\Gamma)} c_x$ generates one constraint for each variable x in the domain of the type environment. There is no need to add a special constraint for this purpose.

The judgment $\Omega, \Gamma, C \vdash_{\mathrm{v}} v : \alpha$ generates the constraint C for the value v. Only the rule for functions (FUNCTION) is presented because the other ones are trivial. The rule creates a new variable for the singleton environment (σ_2) in which the function should be typed, restricts Γ to the set of free variables in the function, demotes them, and ensures that all effects of the function body are collected and attached to the function type.

The notation $\overline{C_i}$ stands for the conjunction of all constraints mentioned in the precondition of the judgment. (For the FUNCTION rule $\overline{C_i} = C_1 \wedge \ldots \wedge C_7$.)

FUNCTION

$$\frac{\begin{array}{c}\alpha_0, \alpha_2, \mu, \mu', \sigma_2 \text{ fresh} \qquad \Gamma' = \Gamma \downarrow \mathrm{fv}(\lambda x.e) \\ C_1 = \mu_s \cap \mu = \emptyset \qquad C_3 = \mu'' \subseteq \mu \qquad C_4 = \bigwedge_{x \in \mathrm{dom}(\Gamma')} \mathrm{Locs}(\Gamma'(x)) \subseteq \mu' \\ \mu_s = \mathrm{dom}(\sigma_2) \qquad C_5 = \bigwedge_{x \in \mathrm{dom}(\Gamma')} \Gamma''(x) =^{\#}_{\mu'} \Gamma'(x) \\ C_2 = \mu' \subseteq \mu \qquad C_6 = \bigwedge_{x \in \mathrm{dom}(\Gamma')} \sigma_1, \Gamma'(x) \lhd \Gamma''(x) \\ \Omega, \Gamma(x : \alpha_2), \sigma_2, C_7 \vdash_{\mathrm{e}} e : \alpha_1 \Rightarrow \mu'', \sigma_1, \Gamma''' \end{array}}{\Omega, \Gamma, \overline{C_i} \vdash_{\mathrm{v}} \lambda x.e : (\sigma_2, \alpha_0 \times \alpha_2) \xrightarrow{\mu} (\sigma_1, \alpha_1)}$$

Fig. 15. Constraint Generation for Values

The judgment $\Omega, \Gamma, \sigma, C \vdash_{\mathrm{e}} e : \alpha \Rightarrow \mu, \sigma, \Gamma$ generates constraints for expressions. Let's start with a simple, but interesting case, the NEW rule. As in the logical system, we need to ensure that the singleton environment does not contain an object at the abstract location ℓ. We generate C_2 for this purpose and ensure with C_1 and C_3 that there is no reference to an singleton ℓ-object in the environment σ or in free variables. The first condition is no constraint, it is already ensured during constraint generation. The READ rule has to do a recursive call and generates a read constraint. The WRITE rule is similar.

The DEMOTE rule has to demote the type environment Γ and pass it to the recursive call. Hence, the constraint generation algorithm creates a set of new type variables together with a new type environment Γ', such that $\mathrm{dom}(\Gamma) = \mathrm{dom}(\Gamma')$. The constraint C_2 ensures that the newly generated type variables are equal to the demoted images of the original ones from Γ.

The FUNCTION CALL rule guarantees that the singleton environments are compatible with the environments in the function type and that the types of the parameters match.

7.3 Constraint Simplification

Unfortunately, constraint simplification has to deal with negative information like $\ell \notin \mu$. Such a constraint is implicitly generated from $\sigma' = \sigma[\ell \mapsto \{\}]$[5] in the constaint generation for `new`, for example. For this reason a simple monotone framework would not work. Hence we combine two monotone frameworks where one is collecting positive information like $\ell \in \mu$ and the other one is collecting the negative information $\ell \notin \mu$. The positive information raises the lower bounds for location variables (and other variables) and the negative information lowers their upper bounds. We obtain an initial upper bound for the location variables by typing the program under a closed world assumption.

Following the constraint generation phase, all constraints are inserted into a work list and subjected to simplicifation. Simplifying a constraint can cause one or more of the following actions:

[5] The map update requires $\ell \notin \mathrm{dom}(\sigma)$.

$$\textsc{New}\quad \frac{\ell \in \mathrm{dom}(\Omega) \qquad C_1 = \sigma =^{\#}_{\ell} \sigma \qquad C_2 = \sigma' = \sigma[\ell \mapsto \{\}] \qquad C_3 = \bigwedge_{x \in \mathrm{dom}(\Gamma)} \Gamma(x) =^{\#}_{\ell} \Gamma(x)}{\Omega, \Gamma, \sigma, \overline{C_i} \vdash_{\mathrm{e}} \mathtt{new}^{\ell} : \mathtt{obj}(@\ell) \Rightarrow \{l\}, \sigma', \Gamma}$$

$$\textsc{Read}\quad \frac{\alpha \text{ fresh} \qquad \Omega, \Gamma, \sigma, C_1 \vdash_{\mathrm{v}} v_1 : \mathtt{obj}(\xi\mu_1) \qquad C_2 = \sigma \vdash_{\mathrm{r}} \xi\mu_1.a : \alpha}{\Omega, \Gamma, \sigma, \overline{C_i} \vdash_{\mathrm{e}} v.a : \alpha \Rightarrow \emptyset, \sigma, \Gamma}$$

$$\textsc{Demote}\quad \frac{\begin{array}{c}\mu_s = \mathrm{dom}(\sigma') \qquad C_1 = \bigwedge_{l \in \mu'} \sigma, \mathtt{obj}(@l) \lhd \mathtt{obj}(\tilde{}l) \\ \sigma' \text{ fresh} \qquad C_2 = \bigwedge_{x \in \mathrm{dom}(\Gamma)} \Gamma'(x) =^{\#}_{\mu'} \Gamma(x) \qquad \Omega, \Gamma', \sigma', C_3 \vdash_{\mathrm{e}} e : \alpha \Rightarrow \mu, \sigma'', \Gamma'' \\ C_4 = \sigma' =^{\#}_{\mu'} \sigma \qquad C_5 = \mu' \cap \mu_s = \emptyset \qquad C_6 = \mu' \subseteq \mu\end{array}}{\Omega, \Gamma, \sigma, \overline{C_i} \vdash_{\mathrm{e}} \natural^{\mu'} e : \alpha \Rightarrow \mu, \sigma'', \Gamma''}$$

$$\textsc{Function Call}\quad \frac{\begin{array}{c}\alpha, \alpha_0, \mu, \sigma_1 \text{ fresh} \qquad C_1 = \bigwedge_{x \in \mathrm{dom}(\Gamma)} \Gamma(x) =^{\#}_{\mu} \Gamma(x) \\ C_2 = \sigma =^{\#}_{\mu} \sigma \qquad C_4 = \mu_s \cap \mu = \emptyset \qquad \mu_s = \mathrm{dom}(\sigma) \qquad \Omega, \Gamma, \sigma, C_5 \vdash_{\mathrm{v}} v_1 : \alpha_1 \\ \Omega, \Gamma, \sigma, C_6 \vdash_{\mathrm{v}} v_2 : \alpha_2 \qquad C_7 = \mathtt{udf} <: \alpha_0 \qquad C_8 = \alpha_1 = (\sigma, \alpha_0 \times \alpha_2) \xrightarrow{\mu} (\sigma_1, \alpha)\end{array}}{\Omega, \Gamma, \sigma, \overline{C_i} \vdash_{\mathrm{e}} v_1(v_2) : \alpha \Rightarrow \mu, \sigma_1, \Gamma}$$

Fig. 16. Constraint Generation for Expressions

1. Create a new constraint,
2. Raise the lower bound of a variable,
3. Lower the upper bound of a variable,
4. Remove the constraint from the constraint set.

For space reasons, we only consider a few select simplification rules. The technical report [13] contains all of them. Let's have a look at the constraint simplification rule for $l \in \mu$:

$$l \in \mu \qquad \rightarrow \qquad \mathtt{delete}, \underline{\mu} := \underline{\mu} \cup \{l\}$$

Executing this rule raises the lower bound of μ (denoted by $\underline{\mu}$) by adding l. Afterwards $\underline{\mu}$ contains all the information that was expressed by the constraint, so the constraint itself is deleted.

The operation $\cup$ for a location variable does some additional things. First, it checks if l is not contained in the upper bound of μ, denoted $l \notin \overline{\mu}$. If that is the case, then the constraint $l \in \mu$ is not solvable because it contradicts the already computed upper bound. Second, if the lower bound of μ changes (viz., if $\underline{\mu}$ did not contain l before the update), then every constraint which depends on μ is added to the work list.

Another, more complicated example is the simplification of a read constraint. Let's assume we have

$$\sigma \vdash_{\mathrm{r}} \xi\mu_1.a : \alpha$$

and that $\underline{\mu_1} = \{l_1\}$ and $\overline{\mu_1} = \{l_1, l_2, l_3\}$, the precision variable $\xi = @$, $\sigma(l_1) = o_1$, $o_1 = []$. The constraint simplification finds that ξ is equal to @ and that implies

that only one location is allowed for μ_1. Because the lower bound of μ_1 contains one element, simplification sets $\mu_1 = \{l_1\}$ and lowers the upper bound of μ_1 to $\overline{\mu_1} = \{l_1\}$. After this change of the state of μ_1 every constraint that depends on μ_1 is added to the work list. In particular, the read constraint is visited once more.

The next visit notices that μ is equal to a location, and that the precision variable is set. Hence, $\sigma(l)$ contains information about the shape of the object. Reading the property a of the object enforces that the object has a property that is a subtype of α. Hence, simplification extends $o_1 = [a \mapsto \alpha_a]$ and generates a new constraint $\alpha_a <: \alpha$. If one of these operations is not allowed, for example, because the upper bound of o_1 ensures that $a \notin \mathrm{dom}(o_1)$, a False constraint is generated. Of course, the newly generated subtype constraint is added to the work list, as well as each constraint that depends on o_1.

Of course, the algorithm considers many other cases but space does not permit to discuss them here. Experience with our prototype implementation indicates that the combininations of two monotone frameworks enables the algorithm to collect sufficient information to infer a valid typing.

Currently, the implementation is not optimized for speed. For example, it propagates the structure of the most recent heap to every program point. A possibility to make the algorithm faster by reducing the amount of data computed is to use lazy propagation like Foster and coworkers [10].

8 Related Work

The idea of using allocation points for abstracting heap structures and the per-program-point approximation of the heap are due to Jones and Muchnick [16].

The notion of strong update is due to Chase and coworkers [6]. Their analysis relies on complicated rules involving a per-program-point "storage shape graph", and no correctness argument is given. The present work reduces the storage shape information to the (per-program-point) singleton environments, and it comes with a correctness proof.

Our type system is related to must-alias analysis [1] and uniqueness typing [5]. Indeed, while the imprecise type $\texttt{obj}(\tilde{}L)$ expresses may-alias information, the precise type $\texttt{obj}(@\ell)$ expresses that all variables of this type refer to the same object. The information conveyed is different from uniqueness, which guarantees that some variable holds the **only** reference to a heap object. A commonality to Altucher and Landi's approach [1] is that their object names also refer exactly to the most recently allocated object of a particular allocation point. Thus, we can answer a question raised in previous work [14] affirmatively: the idea of their approach can be extended to a higher-order language.

Balakrishnan and Reps [3] present an analysis called "Recency-Abstraction for Heap-Allocated Storage". Their goal is to obtain precise abstractions for pointers to objects in executables. They exploit this information to optimize dynamic dispatch in C++ binaries to static function calls where possible.

Smith, Walker, and Morrisett [22, 25] consider alias types as an extension of linear types for typing low-level languages. Their calculus models object initialization with type changing assignments to heap records. Alias types separate pointers types from the actual store contents. A pointer has a singleton type $\texttt{ptr}(l)$, where l stands for a store location. They rely on existential quantification to specify recursive data structures. Explicit pack and unpack operations are needed for existentials and for recursive types. In contrast, our precise pointer type $\texttt{obj}(@\ell)$ is a singleton type standing for a location *at a particular program point*. Our $\texttt{obj}(\tilde{}L)$ type has an existential-type flavor and it can model recursive data structures where demotion corresponds to packing. Finally, the alias types system is a prescriptive, explicitly typed calculus with decidable type checking whereas our calculus is descriptive, implicitly typed, and has type inference.

Cqual [10] is a tool for specifying and inferring flow-sensitive type qualifiers in C programs, which is related to typestate inferencee. Cqual first performs a flow-insensitive type, alias, and effect analysis. In a second phase, it infers linearities, which it uses to perform strong updates on assigned type qualifiers. Cqual differs in a number of technical aspects from our work, one point being that our store abstraction can always handle one reference per abstract store location exactly (linearly in Cqual terminology) as well as many summary reference at the same time whereas Cqual classifies a store location as either linear or non-linear. Cqual's use of polymorphism is similar to our notion of store splitting, which is explained in our technical report.

Fähndrich and Xia's delayed types [11] also provide a means of treating object initialization. An object with a delayed type does not have to fulfill its invariants, yet. The example in their paper is type checking not-null types. Our calculus could be put to use for a similar analysis; at present, with delayed types the programmer provides an explicit boundary when the invariants must hold, whereas our calculus tracks exact information about an object as long as possible and reverts to less precise summary information when unavoidable. We expect that the recency attribute holds sufficiently long to cover the initialization phase, but further investigation is required to confirm this expectation.

Kehrt and Aldrich [17] explore an imperative variant of Abadi and Cardelli's object calculus with delegation, linear object types, and linear methods. As long as objects have a linear type, their method suite and delegatee can be changed as typical in an initialization phase. Later on, the programmer can drop linearity of an object at the price of making it immutable. Recency can achieve similar objectives without requiring the object to be linear and without making it immutable. The object loses its special status only when the next object is allocated at the same abstract location.

Previous work involving the second author [14] defines the notion of singleness. Singleness is a property of a variable that implies must-alias information. A variable x is single at expression e if all executions leading to e cause all bindings for x to be identical. As this property is variable-oriented and not connected to the notion of recent allocation, it is quite incomparable to the present work, although the results could be used for similar purposes.

Might and Shivers [19] extend control flow analysis with abstract garbage collection and abstract counting, the abstract counterpart of reference counting. A superficial look suggests a relation to $\mathcal{RAC}$, which cannot be substantiated: An object with a precise reference type $@\ell$ does not imply that the same object would have an abstract count of one. Indeed, the current summary heap might contain an arbitrary number of reachable imprecise references of type $\tilde{}\ell$. $\mathcal{RAC}$ does not support any notion of garbage collection. The computation of linearities in the work of Foster and coworkers [10] has properties very similar to abstract counting.

Anderson and coworkers [2] define a type system for JavaScript that comes with a type inference algorithm. Their type system is based on an extension of record types by a definedness indicator on each record component. The latter distinguish whether a record component is definitively initialized or whether it may be uninitialized. Compared to our work, they do not model type change and they do not consider prototypes. Our implementation can type check all examples in that work. We have not been able to get meaningful results from their implementation, which precludes further comparison.

A similar idea is the basis for Qi and Myers' masked types [21]. Their typestate-based system tracks the initialization of objects in a Java core language. It sidesteps the need for aliasing control by reverting to the monotonic property "maybe initialized". It requires sophisticated hand-annotated types for dealing with cyclic data structures. Our system handles cyclic initialization without annotation (as in Fig. 1(b)). It cannot elide (local) aliasing control because general type change is not monotonic, but required for analyzing scripts.

In previous work, the second author has proposed a type system for JavaScript [23]. The prime objective of that system was the study of a dynamic typing approach that enables to detect the "undesirable conversions" discussed in the introduction with high precision. The present work is complementary in that our previous work did not support flow-sensitive types.

Jensen and coworkers [15] have built a static analyzer for the entire JavaScript language. This system is based on abstract interpretation and relies on recency, context sensitivity, and some other techniques to obtain precise results. That work is complementary because it is a practical implementation which is not supported by formal proof. Moreover, it suffers from the restriction of all abstract interpretation-based systems that it only affirms that a particular program does not misbehave on a given set of inputs. Thus, unlike our present system, it cannot compute a function type that describes the set of admissible inputs.

Some researchers treat scripting language typing from the prescriptive point of view. A prominent example is the work on typed Scheme [24]. Their goal is to enrich untyped languages gradually with type assertions and contracts and thus transition over time to an explicitly typed language.

Bono and Fisher [4] consider an imperative object calculus with object extension and encapsulation. The goal of their calculus is to demonstrate that classes and inheritance can be implemented in an object-based calculus if it provides extension and encapsulation. Their calculus distinguishes prototypes (extensible

and overridable records without subtyping) from objects (read-only records with subtyping). Overwriting of fields or methods with a different type is not possible. They define a sound and complete typing algorithm. This work extends Fisher's thesis [9], which considers a functional calculus with similar features and with a may-type mechanism, but which does not include a typing algorithm.

9 Conclusion

A type system that incorporates recency abstraction automatically identifies an initialization phase at the beginning of the lifetime of an object. During initialization, the object's type is flow-sensitive and is amenable to precise type change via strong update. The initialization phase ends when another object is created at the same allocation point. Afterwards, the object's type is a flow-insensitive summary of the initialized state and all further updates.

These features make the recency-based type system well suited for analyzing scripting languages. It gives satisfactory results on examples in the JavaScript core language considered in this paper and it is amenable to accurately track JavaScript's prototype objects. While the analysis results are more conservative than in an abstract interpretation-based approach, they are also more general because the typing approach allows the stand-alone analysis of libraries.

Besides the full technical details and proofs, the accompanying technical report [13] contains some worked out extensions to the theory, most notably the treatment of conditionals, prototypes, and store splitting, which provides a notion of polymorphism with respect to locations. We are currently extending the implementation with context-sensitivity, which is straightforward to add, to improve the analysis of factory methods. We are also interested in extending the implementation with the goal to apply it to real-life JavaScript programs.

References

1. R. Z. Altucher and W. Landi. An extended form of must alias analysis for dynamic allocation. In *Proc. 1995 ACM Symp. POPL*, pages 74–84, San Francisco, CA, USA, Jan. 1995. ACM Press.
2. C. Anderson, P. Giannini, and S. Drossopoulou. Towards type inference for JavaScript. In *19th ECOOP*, volume 3586 of *LNCS*, Glasgow, Scotland, July 2005. Springer.
3. G. Balakrishnan and T. W. Reps. Recency-abstraction for heap-allocated storage. In K. Yi, editor, *SAS*, volume 4134 of *LNCS*, pages 221–239. Springer, 2006.
4. V. Bono and K. Fisher. An imperative, first-order calculus with object extension. In E. Jul, editor, *12th ECOOP*, volume 1445 of *LNCS*, pages 462–497, Brussels, Belgium, July 1998. Springer.
5. J. Boyland, J. Noble, and W. Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP '01: Proc. 15th European Conference on Object-Oriented Programming*, pages 2–27, London, UK, 2001. Springer-Verlag.
6. D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proc. PLDI '90*, pages 296–310, White Plains, NY, USA, June 1990. ACM.

7. D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, 2008. 170 pages.
8. ECMAScript Language Specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, Dec. 1999. ECMA International, ECMA-262, 3rd edition.
9. K. Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Standford University, 1996. Stanford CS Technical Report STAN-CS-TR-98-1602.
10. J. S. Foster, T. Terauchi, and A. Aiken. Flow-sensitive type qualifiers. In PLDI 2002 [20], pages 1–12.
11. M. Fähndrich and S. Xia. Establishing object invariants with delayed types. In *Proc. 22nd ACM Conf. OOPSLA*, pages 337–350, Montreal, QC, CA, 2007. ACM Press, New York.
12. D. Gifford and J. Lucassen. Integrating functional and imperative programming. In *Proc. 1986 ACM Conf. on Lisp and Functional Programming*, pages 28–38, 1986.
13. P. Heidegger and P. Thiemann. Recency types for scripting languages. Universität Freiburg, July, 2009. `http://proglang.informatik.uni-freiburg.de/JavaScript/appendix.pdf`.
14. S. Jagannathan, P. Thiemann, S. Weeks, and A. Wright. Single and loving it: Must-alias analysis for higher-order languages. In L. Cardelli, editor, *Proc. 25th ACM Symp. POPL*, pages 329–341, San Diego, CA, USA, Jan. 1998. ACM Press.
15. S. H. Jensen, A. Møller, and P. Thiemann. Type analysis for JavaScript. In *Proc. 16th International Static Analysis Symposium, SAS '09*, volume 5673 of *LNCS*. Springer-Verlag, Aug. 2009.
16. N. D. Jones and S. S. Muchnick. Flow analysis and optimization of Lisp-like languages. In *Proc. 6th ACM Symp. POPL*, pages 244–256. ACM Press, 1979.
17. M. Kehrt and J. Aldrich. A theory of linear objects. In *FOOL 2008*, San Francisco, CA, USA, Jan. 2008. `http://fool08.kuis.kyoto-u.ac.jp/kehrt.pdf`.
18. S. Lebresne, G. Richards, J. Östlund, T. Wrigstad, and J. Vitek. Understanding the dynamics of JavaScript. In *International Workshop on Script to Program Evolution (STOP)*, Genova, Italy, July 2009.
19. M. Might and O. Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In J. Lawall, editor, *Proc. ICFP 2006*, pages 13–25, Portland, Oregon, USA, Sept. 2006. ACM Press, New York.
20. *Proc. 2002 PLDI*, Berlin, Germany, June 2002. ACM Press.
21. X. Qi and A. C. Myers. Masked types for sound object initialization. In B. Pierce, editor, *Proc. 36th ACM Symp. POPL*, pages 53–65, Savannah, GA, USA, Jan. 2009. ACM Press.
22. F. Smith, D. Walker, and J. G. Morrisett. Alias types. In G. Smolka, editor, *Proc. 9th ESOP*, volume 1782 of *LNCS*, pages 366–381, Berlin, Germany, Mar. 2000. Springer.
23. P. Thiemann. Towards a type system for analyzing JavaScript programs. In *Proc. 14th ESOP*, volume 3444 of *LNCS*, pages 408–422, Edinburgh, Scotland, Apr. 2005. Springer.
24. S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In P. Wadler, editor, *Proc. 35th ACM Symp. POPL*, pages 395–406, San Francisco, CA, USA, Jan. 2008. ACM Press.
25. D. Walker and G. Morrisett. Alias types for recursive data structures. In R. Harper, editor, *Proc. ACM Workshop Types in Compilation (TIC'00)*, volume 2071 of *LNCS*, pages 177–206, Montréal, Canada, Sept. 2000. Springer.
26. A. Wright and M. Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.