Recency Types for Dynamically-Typed, Object-Based Languages

Strong Updates for JavaScript

Phillip Heidegger Peter Thiemann

Universität Freiburg, Germany

{heidegger,thiemann}@informatik.uni-freiburg.de

Abstract

Object-based languages with dynamic type systems are popular because they accelerate the development of short programs. As larger programs are built with those languages, static analyses become important tools for detecting programming errors.

We define such an analysis as a type system for an imperative object-based calculus and prove its type soundness. The calculus models essential features of the JavaScript language, as an example of a typical and widely used dynamically-typed, object-based language. The model includes objects as property maps, type change during object initialization, and precise support for JavaScript's prototype mechanism.

As a more general technical contribution, our work demonstrates that the idea of recency abstraction can be transferred from abstract interpretation to a type system. We model recency information with a notion of precise object pointers that enables strong, type changing updates of object types during a generalized initialization phase. The same precise object pointers allow for an accurate treatment of the prototype mechanism. Unlike linear types, precise object pointers can be nested and mixed arbitrarily with ordinary, imprecise object pointers in the type of a data structure.

1. Introduction

Modern web applications are abundant with dynamic features like animations, pop-down menus, drag-and-drop, and wysiwyg-editors, just to name a few. While there are a number of toolkits at various degrees of abstraction to develop such applications (GWT [15], dojo [7], scriptaculous [23], and many more), there is still a number of significant applications written directly in JavaScript.

Despite tremendous efforts and progress in IDEs and debuggers, the development of such native JavaScript applications still suffers from numerous problems. One source of these problems is the weak, dynamic type system of JavaScript and its prototype-based nature.[1] Another host of problems is introduced by different implementations of the browser API by different vendors. Although the latter problems are equally severe from a practical point of view, this paper concentrates on the typing problems. As many other scripting languages (like Python, Ruby, PHP, Lua, Perl, Tcl) are dynamically typed and object based, any progress in analyzing JavaScript programs can find fruitful application to the analysis of properties for other scripting languages as well.

1.1 Problems with Typing JavaScript

Weak, dynamic typing means in JavaScript that values are subject to extensive type conversion rules at run time. While each value is created at a certain type and keeps the type during its lifetime, a use of the value can convert it implicitly into almost any other type. Only a few of these conversions are forbidden and yield run-time errors.[2] Of the remaining legal conversions, quite a few are counterintuitive or outright undesirable [27]. Thus, an analysis should be able to generate warnings for conversions that give unexpected results or have unexpected side effects.

As in many scripting languages, an object in JavaScript is represented by a property map that associates a value to a property name. If this value happens to be a function, then the property is a method, otherwise it is a field of the object. Creating an object amounts to allocating a new, empty property map and running a constructor method on this map. The constructor method enters values into the property map, thus creating fields and methods of the object. However, the creation of fields and methods is not restricted to the constructor, but it may happen at any time during execution.

An entry in a property map does not obey any type discipline. The type of the value assigned to a property can change with every assignment. Moreover, as long as a property is not yet assigned, accessing it returns the value `undefined`, which is a regular JavaScript value. This feature makes it very hard for a flow-insensitive type system to infer a precise type for a JavaScript object: each property is initially `undefined`, so that the type of a property would have to be a union type which always includes an `undefined` participant.

While such a type system has its merits, its typings yield a very crude approximation of the true type of a property. One particular application of such a type system is the test whether the conversion of a field value to some type fails or may give unexpected results. This test raises too many false positives if the type of the property is not sufficiently precise.

An essential feature of JavaScript is its prototype mechanism. It implements a delegation mechanism, which simulates some notion of single inheritance. Each object has an associated prototype

[1] This paper does not address features or problems of the upcoming 4th edition (ES4) of the language [9]. Instead, it discusses the JavaScript language as it is shipped with current browsers, which is roughly in line with the ECMAScript standard [8]. Due to the installed browser base, this language will remain in the focus of web developers for some years to come. The core features of our analysis, in particular the tracking of prototype links, will also be useful for analyzing ES4 programs.

[2] For example, only a function can be converted to a function and the value `undefined` cannot be converted to an object.

FOOL '09. 24 January 2009, Savannah, Georgia, USA.
ACM.

```
1 var proto = { a : "foo" };
2 function Make () {}; Make.prototype = proto;
3 var obj = new Make (); obj.b = "gnu";
4 obj.b; // returns "gnu"
5 obj.a; // returns "foo"
```

Figure 1: Using a prototype in JavaScript. Line 1 creates a new object with property `a` set to `"foo"` and binds it to variable `proto`. Line 2 defines a function `Make` with no arguments and no body; it always returns `undefined` and sets the `prototype` property of function `Make` to `proto`. Line 3 creates a new object using `Make` as a constructor. Because of `Make`'s `prototype` property, the resulting object `obj` is internally linked to `proto` as its prototype object. The remaining lines write and read the `b` and `a` properties of `obj`.

object that is set by the constructor. The prototype object provides default values for properties of the original object. If a property is not present in the original object, then JavaScript recursively attempts to access the property in the prototype object until it hits an undefined prototype. However, write accesses directly affect the original object, not the prototype.

For example, let `proto` be an object with property `a` defined to be `"foo"` and `obj` be an object with prototype `proto` and property `b` set to `"gnu"`, as defined in Fig. 1.[3] Accessing `obj.b` immediately returns the value of the `b` property whereas `obj.a` first delegates the property access to the prototype object `proto`, which has the property `a` and directly returns its value.

1.2 Desiderata for Typing JavaScript

In a dynamically-typed language, each update of a property may change the type of the property in the object. A type analysis can keep up with this type change in two ways. Either the analysis is flow insensitive and assigns a summary type to the property (weak update) or the analysis is flow sensitive and assigns different types to the property at different points in the program (strong update). The exact choice of summary type can range from a simple top type to a sophisticated union type.

The type-based analysis that we are proposing attempts to provide the best of both worlds. It targets the typical programming pattern where the programmer allocates a number of objects and then performs some initialization on these objects. Usually, no further properties are defined after the initialization and the type of the properties rarely changes. Hence, an analysis should provide for a flow sensitive initialization phase for each object and then fall back to flow insensitive typing.

In a prototype-based language, the analysis of property accesses requires accurate handling of prototype information. In most circumstances, there is exactly one prototype object for each kind (or class) of objects in a program. Prototype objects are created, filled with default values and methods, and then never changed or recreated anymore. Given these properties of prototype objects, a type-based analysis should be able to answer questions like "does this object have a prototype?" and "does the prototype have this property?" precisely.

1.3 Contributions

- We identify recency information as the key to identify the initialization phase of an object and to provide proper handling of prototype objects. An object is recent as long as no further object has been allocated at the same `new` expression. If recency information is part of a type (an abstraction), then the type (abstraction) of a recent object can be subject to strong updating.
- Recency has been discovered and formalized in the context of abstract interpretation for a first-order imperative language (see Sec. 2). We show that recency can also be expressed with a type system by defining a recency-aware calculus. This calculus has objects and first-class functions, thus our work extends the notion of recency to object-based and higher-order languages. The calculus can be considered a core language of JavaScript.
- We prove type soundness for the recency-aware calculus.

Section 2 provides some background information by explaining the ideas underlying recency abstraction.

Section 3 informally rephrases recency abstraction in terms of a type system and provides motivating examples. The subsequent Section 4 contains the formal definitions for the recency-aware calculus (syntax, dynamic semantics, static semantics).

Next, we establish the metatheory of the recency-aware type system via a syntactic type soundness proof in Section 5. The techniques used are standard but involved because recency-awareness requires a novel way of setting up the operational semantics including a non-standard substitution.

Section 6 considers extensions, Section 7 discusses related work, and Section 8 concludes.

2. Recency Abstraction

Balakrishnan and Reps [3] present an analysis that they call "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.

Their general framework is abstract interpretation. They start off with the standard approach where the analysis associates each pointer-typed variable with a set of object creation sites. The abstract interpretation of a property access then yields the least upper bound of the abstract values associated with the property at each creation site.

Unfortunately, even if the analysis associates exactly one creation site with a pointer variable, the results returned by abstract property accesses are still summaries (*e.g.*, least upper bounds) over the property values of many concrete objects at many different times. Thus, if the property `a` of an object created at program point ℓ gets assigned an integer in one place and a string in another place in the program, an abstract program access will likely return an uninformative abstract value that approximates both, an integer and a string. This situation may occur even if there is no program run such that an object first contains an integer in its property `a` and then a string (or vice versa).

The idea of recency abstraction is a simple, but ingenious variation of this theme. For each creation site ℓ, the analysis keeps track of two abstractions, one for the object most recently allocated at ℓ and another for all older objects allocated at ℓ. In consequence, the abstraction of the most recently allocated object is very precise. In fact, the concrete value can be used as long as there are no intervening conditionals. In any case, the most recent abstraction represents exactly one concrete object at run time, thus the abstract interpreter can treat any property updates to this object as **strong updates**, hence keeping the abstraction of this object as precise as possible. For the same reason, it is straightforward to track aliasing for the most recently allocated object.

Balakrishnan and Reps [3] implement recency abstraction by duplicating the abstract heap into one component which only contains the abstraction of the most recently allocated object of each

[3] The code is not straightforward because the prototype property is an internal property which is not directly assignable. Instead, creating an object with a constructor function (`Make` in the example), on which the `prototype` property is set, establishes the internal prototype property on the new object.

```
1 function f() { return new Object(); /*Location 0*/ }
2 var o1 = f(); /*Location 1*/
3 o1.a = "foo"; o1.b = "gnu"; /*Location 2*/
4 var o2 = f(); o2.a = "moo"; /*Location 3*/
5 o2.a = 5; o1.a = 42; /*Location 4*/
```

Figure 2: Example for recency abstraction. Line 1 defines a function `f` that always returns a new, empty object. Lines 2-3 create the object `o1` by calling `f` and set some of its properties. Line 4 does the same for object `o2` and line 5 sets the `a` property of `o2` and `o1`.

creation site and another component that contains the summary abstraction for all other objects of that creation site. Accordingly, abstract pointer values come in two guises, one that points into the most-recent heap and another that points into the summary heap.

In our terminology, the abstraction associated with a variable that holds an object allocated at location 0 is either a *precise pointer* `@0` or an *imprecise pointer* $^\circ\{0\}$ (in general, there can be a set of creation sites in an imprecise pointer). To dereference a precise pointer, the most-recent heap maps each creation site to an abstract object, where each property contains an abstract value, which may again be a precise or imprecise pointer. To dereference an imprecise pointer, the summary heap maps each creation site to an abstract object, where each property contains an abstract value.

Fig. 2 contains a sample of JavaScript code to illustrate the idea of recency abstraction. The abstraction of the single creation site in the program is its location 0 in function `f`. At location 1, the program has created one object in variable `o1` and all its properties are `undefined`. As this object is the one most recently created at 0, its abstraction reflects this empty object exactly. At location 2, the program has updated the properties `a` and `b` of `o1` with strings. As `o1` still contains the most recently allocated object with location 0, the abstraction at this program point can still represent the object {`a="foo", b="gnu"`} exactly.

The creation of the second 0-labeled object after 2 degrades `o1` to a non-recent object and merges the value of `o1` with the mapping for inexact 0-labeled objects. At 3 the state of the abstract interpreter is `o1 =` $^\circ$`0`, `o2 = @0` where `@0` is associated with the object {`a="moo"`} in the most-recent heap and $^\circ$`0` is associated with the abstract object {`a:string, b:string`} in the summary heap.

The next two updates concern one precise and one imprecise pointer. The precise one induces a strong update in the abstract interpreter and changes the value of the `a` property of `o2`. The imprecise one assigns an integer to the property `a` of an object that used to have a property `a` of type string. The type for 0 in the summary heap must cater for both possibilities and lifts the type of `a` to `top`. At 4, the final state of the abstract interpreter is `o1 =` $^\circ$`0`, `o2 = @0` where `@0` is associated with {`a=5`} and $^\circ$`0` is associated with {`a:top, b:string`}.

The example illustrates a key issue in recency abstraction. Whenever a new object is created at creation site ℓ, the previous most recent abstraction must be moved from the most-recent heap into the summary heap. Its abstract information is merged with the information of all the other ℓ-objects and its precise pointer demoted to an imprecise pointer. Then, the new object is created in the most-recent heap with a precise pointer that references precise abstract information. The abstract interpreter can easily perform this demotion by examining and changing abstract values and by updating the summary heap at ℓ to the least upper bound of the old summary heap cell ℓ and the most-recent heap cell ℓ.

3. Recency Typing, Informally

This section examines the idea of recency abstraction from a typing perspective. There are three key points. First, recency distinguishes between the most recently allocated object and the older ones and it maintains two abstractions, a most recent one and a summary one. Second, the abstraction for the most recently allocated object is subject to strong update. Third, while an abstract interpreter can move a reference from the most recent heap to the summary heap at any point in the program, a type system may have to apply restrictions to stay tractable.

These key points need to be reflected in the design of the operational semantics and the type system. While this section concentrates mainly on the type system it also touches on issues of the operational semantics.

In addition to the typing environment Γ, the typing judgments mention two further environments, the summary environment Ω and the most-recent environment Σ. They keep the summary type and the most recent type for each abstract location ℓ. An abstract location is an arbitrary marker attached to a `new` expression, which is not necessarily unique. It abstracts the set of store locations returned by the `new` expression. As the summary abstraction only supports weak updates, the summary environment is globally the same for all program points. In contrast, the most recent environment may change in each expression and is thus threaded through by the typing judgment.

$$\Omega, \Sigma, \Gamma \vdash_e e : t \Rightarrow L, \Sigma', \Gamma'$$

The judgment also threads the typing environment, because the demotion of an object from the most recent heap to the summary heap also changes its type. The component L is an effect and is explained with the discussion of functions at the end of section 3.2. There is no separate discussion of methods: They are modeled as functions stored in object properties.

The language of expressions is not JavaScript, but rather a suitable core language with objects and first-class functions. One particular difference is the `new` operator, which just creates an empty object with no properties. To obtain the effect of a JavaScript constructor invocation $x = \texttt{new} f(\dots)$, the following code fragment would be a first approximation.[4]

$$\begin{array}{l}\texttt{let } x = \texttt{new in}\\ \texttt{let } \_ = (x.c = f) \texttt{ in}\\ \texttt{let } \_ = x.c(\dots) \texttt{ in}\end{array}$$

3.1 Object Types

The type of an object is either a precise object type $\texttt{obj}(@\ell)$ or an imprecise object type $\texttt{obj}(\tilde{}L)$. A precise object type is very similar to a singleton reference type [25]. It references one particular entry in the most recent environment and can keep track of a limited amount of aliasing. Here is an example, which first constructs an empty object, copies its pointer to y, defines a property a through y, and finally reads a through x:[5]

$$\begin{array}{l}\texttt{let } x = \texttt{new}^\ell \texttt{ in}\\ \texttt{let } y = x \texttt{ in}\\ \texttt{let } z = (y.a = 5) \texttt{ in } x.a\end{array}$$

The typing for the final subterm $x.a$ of this expression is

$$\begin{array}{l}\Omega, \Sigma, \Gamma \vdash_e x.a : \texttt{int} \Rightarrow \emptyset, \Sigma, \Gamma\\ \Sigma = [\ell \mapsto [a \mapsto \texttt{int}]]\\ \Gamma = [x : \texttt{obj}(@\ell), y : \texttt{obj}(@\ell), z : \texttt{udf}]\end{array}$$

where the typing environment Γ indicates that both, x and y, refer to the same object in the most recent heap at ℓ and the most recent

[4] This code fragment serves as an incomplete illustration. It ignores the return value of the method and it does not install a prototype.

[5] The examples make liberal use of base types like `int`, `string`, and `bool` as well as obvious syntax for introducing values of these types, although the formal calculus does not encompass them.

environment Σ indicates that ℓ refers to an object which has an a property of type `int` and which is otherwise undefined. The summary environment Ω does not matter in this derivation.

Precise object types enable strong update in the most-recent environment as the next example demonstrates.

$$\begin{array}{l}\texttt{let } x = \texttt{new}^{\ell} \texttt{ in}\\ \texttt{let } y = x \texttt{ in}\\ \texttt{let } z = (y.a = 5) \texttt{ in}\\ \texttt{let } w = (x.a = \text{"}\texttt{crunch}\text{"}) \texttt{ in } y.a\end{array}$$

The typing for the final subterm $y.a$ of this expression is

$$\begin{array}{l}\Omega, \Sigma, \Gamma \vdash_e y.a : \texttt{string} \Rightarrow \emptyset, \Sigma, \Gamma\\ \Sigma = [\ell \mapsto [a \mapsto \texttt{string}]]\\ \Gamma = [x : \texttt{obj}(@\ell), y : \texttt{obj}(@\ell), z : \texttt{udf}, w : \texttt{udf}]\end{array}$$

The indirection of object types through the most recent environment leaves the type of both object pointer unchanged. But the dereferencing through the most-recent environment changes the types of the properties visible through either pointer.

This phenomenon explains why an object type must not contain more than one precise pointer. If an object type of the form $\texttt{obj}(@\ell_1, @\ell_2)$, say, were permitted, then a strong update would be unsound. To see why, assume that the update at run-time affects the object abstracted by ℓ_1 and suppose there is a further reference to the other object abstracted by ℓ_2. As a strong update would change the type of both objects, the type for the unchanged ℓ_2 object would become invalid for the run-time object.

The summary environment and imprecise object pointers kick in as soon as multiple objects are created at the same abstract location. In the example terms, there are multiple `new`s with the same label to keep the examples simple. To obtain good precision in practice, each `new` in a program should have a unique label.

$$\begin{array}{l}\nabla^{\ell}\texttt{let } x = \texttt{new}^{\ell} \texttt{ in let } \_ = x.a = 42 \texttt{ in}\\ \nabla^{\ell}\texttt{let } y = \texttt{new}^{\ell} \texttt{ in let } \_ = y.a = \text{"}\texttt{flush}\text{" in}\\ \nabla^{\ell}\texttt{let } z = \texttt{new}^{\ell} \texttt{ in let } \_ = z.a = \texttt{true in}\\ x.a\end{array} \tag{1}$$

Here is the typing for the final $x.a$ expression

$$\begin{array}{l}\Omega, \Sigma, \Gamma \vdash_e x.a : \top \Rightarrow \emptyset, \Sigma, \Gamma\\ \Omega = [\ell \mapsto [a \mapsto \top]]\\ \Sigma = [\ell \mapsto [a \mapsto \texttt{bool}]]\\ \Gamma = [x : \texttt{obj}(\tilde{}\{\ell\}), y : \texttt{obj}(\tilde{}\{\ell\}), z : \texttt{obj}(@\ell)]\end{array}$$

The typing shows that the summary environment indeed summarizes the types `int` and `string` for the property `a` to the top type $\top$. The imprecise object pointers contain a singleton set $\{\ell\}$ in the example. In general, such a type may contain multiple creation sites. Finally, observe that recency typing would still deduce the typing $z.a$: `bool` if that was the final expression.

It remains to explain the ∇^{ℓ} expressions that appear now in front of the $\texttt{new}^{\ell}$ expressions. These *mask expressions* are not written by the programmer, but automatically inserted in an elaboration phase[6]. They provide an explicit syntactic marker for moving objects from the most-recent heap to the summary heap. They do not act on the expression, but only on the heaps. While masking could be integrated into the typing rules for `new` and function application, introducing an explicit mask expression serves to modularize the type system and its soundness proof. In general, the mask carries a set L of labels of the objects that have to be moved. In (1), the mask appears in front of each $\texttt{new}^{\ell}$ to move the eventual ℓ-object to the summary heap.

In general, a mask expression is applied to each `let` that directly encloses a `new` (as in (1)), a function call, or a method call. The latter mask anticipates demotions that may have to take place inside the function body. The label on the mask for $\texttt{new}^{\ell}$ is exactly the singleton set $\{\ell\}$, whereas the label on the mask for a function call is inferred during type inference. The typing of a mask expression $\nabla^{L} e$ requires the demotion of all L-objects in the environment and in the expression e. Otherwise, the final term $x.a$ of the example would get stuck on execution: Because the object bound to x has been moved to the summary heap by the second mask expression, it cannot be accessed in the most-recent heap, anymore.

[6] The first two examples happen to be type correct even without mask expressions inserted.

3.2 Function Types

This explanation leads to the last and most hairy topic, the treatment of function calls. Let's have a look at an example.

$$\begin{array}{l}\nabla^{\ell}\texttt{let } x = \texttt{new}^{\ell} \texttt{ in let } \_ = (x.a = 42) \texttt{ in}\\ \texttt{let } f = \lambda(\_).x.a \texttt{ in}\\ \nabla^{\ell}\texttt{let } y = \texttt{new}^{\ell} \texttt{ in}\\ \nabla^{\ell}\texttt{let } z = f(0) \texttt{ in } z\end{array} \tag{2}$$

The typing for the final expression z is:

$$\begin{array}{l}\Omega, \Sigma, \Gamma \vdash_e z : \top \Rightarrow \emptyset, \Sigma, \Gamma\\ \Omega = [\ell \mapsto [a \mapsto \top]]\\ \Sigma = [\ell \mapsto [\,]]\\ \Gamma = [\ x : \texttt{obj}(\tilde{}\{\ell\}),\\ \qquad f : (\emptyset, \top \times \top) \xrightarrow{\{\ell\}} (\emptyset, \top),\\ \qquad y : \texttt{obj}(\tilde{}\{\ell\}),\\ \qquad z : \top]\end{array}$$

Naively executing the first line of the program substitutes x with a precise pointer to an object with property `a` set to an integer, also in the body of function f. Executing the second line substitutes f across the object creation in y. The problem is now introduced by the execution of the third line. The mask moves the precise ℓ-object into the summary heap, thus making it imprecise, but the pointer inside f holds on to the precise pointer. Thus, executing the function call in the fourth line gets stuck on the $x.a$ expression because the pointer is now dangling.

Our solution to this problem is the introduction of a *non-standard substitution*. It treats pointers specially by changing a precise pointer into an imprecise one on crossing a function boundary. Adoption of non-standard substitution makes the example expression execute without getting stuck, but it requires the typing rule for functions to demote the types of all free variables inside the function body. The set of the labels of the object types in these variables becomes part of the *effect* L of the function type $(\Sigma_2, t_0 \times t_2) \xrightarrow{L} (\Sigma_1, t_1)$. The effect indicates at which locations the function assumes that precise references for these locations have been moved to the summary heap before calling the function. A small change to the example illustrates the necessity of doing so:

$$\begin{array}{l}\nabla^{\ell}\texttt{let } x = \texttt{new}^{\ell} \texttt{ in let } \_ = (x.a = 42) \texttt{ in}\\ \texttt{let } f = \lambda(\_).x.a \texttt{ in}\\ \nabla^{\ell}(f(0))\end{array} \tag{3}$$

Because the non-standard substitution demotes the precise pointer substituted for x into f, the invocation of f must be preceded by the ∇^{ℓ} to move the object pointed to by x into the summary heap! The ℓ is evident from the effect annotation of f's type

$$(\emptyset, t \times \texttt{int}) \xrightarrow{\{\ell\}} (\emptyset, \texttt{int})$$

The empty environments $\emptyset$ indicate that f does not use the most-recent environment. In general, the leftmost environment in the function type specifies the fragment of the most-recent environment that is passed to the function body whereas the rightmost environment is returned from the function and overrides the respective

parts of the most-recent environment at the callsite of the function. The part $t \times \texttt{int}$ indicates the argument type of the function whereas the `int` is the result type. The argument type is a pair because functions can also serve as methods in which case the first component of the argument type is the type of the receiver object "self". See the formal part in Section 4 for more information on the typing of methods and method invocation.

The following example shows the other use of effects and explains why the typing judgment has to collect the effect of the expression.

$$\begin{array}{l} \triangledown^{\ell}\texttt{let}\ x = \texttt{new}^{\ell}\ \texttt{in let}\ \_ = (x.a = 42)\ \texttt{in} \\ \texttt{let}\ f = \lambda(\_).\texttt{new}^{\ell}\ \texttt{in} \\ \triangledown^{\ell}\texttt{let}\ y = f(0)\ \texttt{in} \\ x.a \end{array} \tag{4}$$

In this expression, the function f allocates an object at the same location as the object pointed to by x, which is still live at the point where f is invoked. The function application replaces the function call by the function's body thus exposing the $\texttt{new}^{\ell}$ expression. As this expression requires masking to ensure correct execution of the expression $x.a$, the effect part of the typing judgment collects the set of all locations in which the expression allocates objects. This effect becomes (the other) part of the effect on the function arrow, so that the type of f is now

$$(\emptyset, t \times \texttt{int}) \xrightarrow{\{\ell\}} ([\ell \mapsto [\,]], \texttt{obj}((@\ell)))$$

Here, the returned most-recent environment describes the newly allocated ℓ-object as everywhere undefined. The function returns a precise pointer to the newly allocated object.

A function can also take a precise object pointer as an argument as long as its location is not in the function's effect.

$$\begin{array}{l} \triangledown^{\ell_1}\texttt{let}\ x = \texttt{new}^{\ell_1}\ \texttt{in let}\ \_ = (x.a = 42)\ \texttt{in} \\ \texttt{let}\ g = \lambda(z).\texttt{let}\ x_1 = \texttt{new}^{\ell_2}\ \texttt{in let}\ \ = (x_1.b = z)\ \texttt{in}\ x_1\ \texttt{in} \\ \triangledown^{\ell}\texttt{let}\ y = g(x)\ \texttt{in} \\ x.a \end{array} \tag{5}$$

The type of g now indicates that an ℓ_1 object is passed precisely into the function body, an ℓ_2 object is created, and an ℓ_2 object which contains the ℓ_1 object in a property is returned from the function call.

$$\begin{array}{l} ([\ell_1 \mapsto [a : \texttt{int}]], t \times \texttt{obj}(@\ell_1)) \\ \xrightarrow{\{\ell_2\}} \\ ([\ell_1 \mapsto [a : \texttt{int}], \ell_2 \mapsto [b : \texttt{obj}(@\ell_1)]], \texttt{obj}((@\ell_2))) \end{array}$$

4. Recency Typing, Formally

This section defines the syntax and semantics of the recency-aware calculus, $\mathcal{RAC}$. After defining some notation, § 4.4 introduces a dynamic semantics tailored for proving type soundness of recency-aware typing and proves some of its invariants in § 4.5. § 4.6 defines the type system proper and § 4.7 extends the static and dynamic semantics with prototypes.

4.1 Syntax

Fig. 3 defines the syntax of expressions in A-normal form [11]. Computations are sequentialized and flattened in our syntax. This choice simplifies the majority of the typing rules because only the rule for `let` has to deal with sequentialization. The price to pay is a slightly more complicated evaluation rule for `let`.

A value is either a variable, a lambda abstraction, `udf` (undefined), or a pointer. A lambda abstraction always takes two parameters because it serves as a first-class function and as a premethod at the same time. When called as a function, the first parameter remains unused. When called as a method, the first parameter becomes the self parameter. Pointer values $(q\ell, i)$ are not present in programs as entered by the programmer. Pointers refer to objects through a heap and arise in intermediate expressions during execution. A pointer may be precise or imprecise which is indicated by $q = @$ or $q = \tilde{}$. Its component $\ell \in \mathsf{Location}$ specifies the creation site of the object. Its i component is a non-negative integer unique among the objects created at ℓ.

$$\begin{array}{rrcl} \mathsf{Value} \ni & v & ::= & x \mid \lambda(y,z).e \mid \texttt{udf} \mid \boxed{(q\ell, i)} \\ \mathsf{Expression} \ni & s & ::= & v \mid v(v) \mid v.a(v) \mid \texttt{new}^{\ell} \mid v.a \mid v.a = v \\ \mathsf{Expression} \ni & e & ::= & v \mid \texttt{let}^{L}\ x = s\ \texttt{in}\ e \mid \boxed{\triangledown^{L} e} \\ \mathsf{Qualifier} \ni & q & ::= & \tilde{} \mid @ \\ \mathsf{Location} \ni & \ell & ::= & l_1 \mid l_2 \mid \ldots \\ \mathsf{Location} \supseteq & L & & \\ \mathsf{Property} \ni & a & & \end{array}$$

Figure 3: Expression syntax. Phrases marked in gray are not written by the programmer. They arise as intermediate steps in the semantics or are inserted automatically by elaboration.

An expression is either a value, a function call, a method call, an object creation, a mask expression, a property read, a property write, or a `let` expression. Except for the `let` and mask expressions, the subexpressions of all other forms are restricted to values. The expression $\texttt{new}^{\ell}$ constructs a new object at creation site ℓ with no defined properties. The mask expression $\triangledown^{L} e$ is discussed in Sec. 3.1. The expressions $v.a$ for reading property a of object v and $v.a = v'$ for defining property a of object v to be v' are standard. The `let` expression is also standard except for the annotation L, which is the effect of the header. The effect annotation guides demotion during substitution (recall that a `let` expression can be desugared to a beta redex).

4.2 Notation

The symbol **N** denotes the set of non-negative integers.

Let $\mathrm{fv}(e)$ denotes the set of free term variables in expression e. The notation $e[(x_1, \ldots x_n) \mapsto (v_1, \ldots v_n)]$ stands for the capture-avoiding, simultaneous substitution of values $v_1, \ldots, v_n$ for the variables $x_1, \ldots, x_n$ in the expression e.

For a mapping m, $\mathrm{dom}(m)$ is the domain of m, *i.e.*, the set of elements x for which $m(x)$ is defined. The notation $m \downarrow X$ restricts the mapping m such that $\mathrm{dom}(m \downarrow X) \subseteq X$. The notation $m \uparrow X$ excludes X from the domain such that $X \cap \mathrm{dom}(m \uparrow X) = \emptyset$. The notation $A \xrightarrow{fin} B$ stands for a partial finite function with domain A and image B.

The *map update* $m' = m\{x \mapsto y\}$ is defined to be $m'(x) = y$ and $m'(x') = m(x')$, for all $x' \neq x$.

If $m \in \mathsf{Property} \to \mathsf{Value}$ is a property map, then property access $m(a)$ is defined by

$$\begin{array}{lcll} \{\}(a) & = & \texttt{udf} & \\ m\{a \mapsto v\}(a) & = & v & \\ m\{b \mapsto v\}(a) & = & m(v) & a \neq b \end{array}$$

For convenience, examples make use of a slightly extended syntax with integer literals of type `int`, arithmetic operations, and syntactic sugar $e_1; e_2$ for a `let` expression $\texttt{let}\ x = e_1\ \texttt{in}\ e_2$ where $x \notin \mathrm{fv}(e_2)$.

4.3 Dynamic Semantics

Fig. 4 defines a straightforward dynamic semantics for $\mathcal{RAC}$. It is a small-step semantics defined on configurations H, e where H is a heap and e is an expression. A heap maps a pair (ℓ, i) of a location

$$
\begin{array}{lcl}
\mathsf{HeapContent} & = & (\mathsf{Property} \xrightarrow{fin} \mathsf{Value}) \\
\mathsf{Heap} & = & (\mathsf{Location} \times \mathbf{N}) \to \mathsf{HeapContent} \\
H & \in & \mathsf{Heap} \\
h & \in & \mathsf{HeapContent} \\
\mathcal{L} & ::= & \Box \mid \mathtt{let}^L\ x = s\ \mathtt{in}\ \Box \mid \triangledown^L \Box
\end{array}
$$

$$
\begin{array}{lllll}
\textsc{S0App} & H, (\lambda(y,x).e)(v) & \to_0 & H, e[(y,x) \mapsto (\mathtt{udf}, v)] & \\
\textsc{S0Let} & H, \mathtt{let}\ x = v\ \mathtt{in}\ e & \to_0 & H, e[x \mapsto v] & \\
\textsc{S0New} & H, \mathtt{new}^\ell & \to_0 & H\{(\ell,i) \mapsto \{\}\}, (\tilde{\ell}, i) & \text{if } (\ell,i) \notin \mathrm{dom}(H) \\
\textsc{S0Read} & H, (q\ell, i).a & \to_0 & H, H(\ell,i)(a) & \\
\textsc{S0Write} & H, (q\ell, i).a = v & \to_0 & H\{(\ell,i)(a) \mapsto v\}, \mathtt{udf} & \\
\textsc{S0MCall} & H, (q\ell, i).a(v) & \to_0 & H, e[(y,x) \mapsto ((q\ell,i), v)] & \text{if } H(\ell,i)(a) = \lambda(y,x).e
\end{array}
$$

$$
\textsc{S0Let}'\quad \frac{H, s \to_0 H', \mathcal{L}[v]}{H, \mathtt{let}^L\ x = s\ \mathtt{in}\ e'' \to_0 H', \mathcal{L}[\mathtt{let}^L\ x = v\ \mathtt{in}\ e'']}
$$

Figure 4: Small-step operational semantics.

and a non-negative integer to some heap content h. A heap content is always a record, *i.e.*, a finite map from property names to values. The following compact notation for updates to a heap H collapses two map updates into one:

$$H\{(l,i)(a) \mapsto v\} := H\{(l,i) \mapsto H(l,i)\{a \mapsto v\}\}$$

Function application is like beta reduction, but it substitutes `udf` for the first argument (the self argument). Reduction of the `let` expression is standard. The `new` expression selects a new, unused memory location for ℓ and stores an empty record in the heap. The read expression extracts the heap content for the argument location and performs a property access on it. Thus, the result of a read expression may be `udf`. The write expression updates the heap content at address (ℓ, i) with the property map, which is in turn updated according to the write expression. The method call expects a function stored in a property. It executes this function by performing a beta reduction applied to the receiving object pointer $(q\ell, i)$ and to the parameter v. The final context rule for `let` expressions forces the header of the let to be evaluated first. If beta reduction yields a value that is wrapped in an $\mathcal{L}$ context, then the rule reestablishes A-normal form by swapping the `let` with the context.

This semantics is close to existing semantics for object-based languages with premethods. However, it is not suitable for proving the soundness of the recency-aware type system defined in Subsection 4.6, because the operational model does not distinguish between the most recently allocated object of a creation site and the objects previously allocated at the same site.

The next subsection rephrases the dynamic semantics to make it more palatable to the soundness proof. The resulting refined dynamic semantics splits the heap in two parts, corresponding to the idea of the recency abstraction, and has an explicit operation to transfer objects from one heap to the other.

4.4 Refined Dynamic Semantics

The refined dynamic semantics also uses small-step operational style. A configuration is a triple (H, H_0, e) with two heaps, the summary heap H and the most-recent heap H_0, and an expression e. The most-recent heap contains —at most— the most recently allocated object for each creation site, the summary heap contains all other objects.

The two heaps are disjoint but share a common address space, $\mathsf{Location} \times \mathbf{N}$. There are two kinds of references, precise references $(@\ell, i)$ that refer to the most-recent heap and imprecise references $(\tilde{\ell}, i)$ that refer to the summary heap. A newly created object always starts out with a precise reference.

There is no restriction on the nesting of precise and imprecise references in data structures. A precise reference can point to an object that contains an imprecise reference and vice versa. The latter facility is key to the ability of $\mathcal{RAC}$ to treat prototypes in a satisfactory manner.

The reduction relation is expected to preserve the following invariant of a configuration:

INV1 For each abstract location, there exists at most one object in the most-recent heap.

The price for splitting the heap in two parts is a more complicated formulation of the reduction rules and the introduction of the mask expression $\triangledown^L e$, as explained in Sec. 3.1.

4.4.1 Elaboration of Mask Expressions

With the refined dynamic semantics, an expression requires *elaboration* before it can be executed without getting stuck. Mask expressions are not present in programs entered by the programmer. An elaboration phase prior to execution wraps a mask expression around every `let` expression that contains a function call, method call, or `new` expression. The L annotation is the effect of the `let` header and thus inferred by the typing algorithm.

4.4.2 Demotion of References

The semantics of the mask expression and the definition of substitution rely on an operation $e^{\natural L}$ that *demotes* precise references for locations L in expression e to imprecise ones. Demotion extends homomorphically to heaps and heap contents, which are treated as mappings. If L is omitted, then $L = \mathsf{Location}$.

$$
\begin{array}{lcl}
x^{\natural L} & = & x \\
(\lambda(y,x).e)^{\natural L} & = & \lambda(y,x).e \\
\mathtt{udf}^{\natural L} & = & \mathtt{udf} \\
(q\ell,i)^{\natural L} & = & \begin{cases} (\tilde{\ell}, i) & \text{if } \ell \in L \\ (q\ell, i) & \text{if } \ell \notin L \end{cases} \\
H^{\natural L} & = & \boldsymbol{\lambda}(\ell,i).H(\ell,i)^{\natural L} \\
h^{\natural L} & = & \boldsymbol{\lambda}a.h(a)^{\natural L} \\
(H, H_0)^{\natural L} & = & H^{\natural L} \cup H_L, H_0^{\natural L} \backslash H_L \\
 & & \text{where } H_L = H_0^{\natural L} \downarrow \{(\ell,i) \mid \ell \in L, i \in \mathbf{N}\}
\end{array}
$$

By invariant **INV-LAM**, which is stated and proved in Sec. 4.5, demotion need not proceed into the body of a lambda abstraction. The last line defines the demotion on a pair of a summary heap and a most-recent heap to include the movement of objects from the latter to the former. H_L is the subset of the most-recent heap—after demotion of its objects according to L—containing those objects which are stored in locations involving L. These objects are transferred to the summary heap and at the same time removed from the most-recent heap (both after demotion).

4.4.3 Substitution

Substitution is the second major operation needed to define the refined semantics. Unfortunately, the presence of precise and imprecise references requires a non-standard notion of substitution, as discussed in Sec. 3.2.

The required notion of substitution of a variable x by a value v in e, $e\{x \Rrightarrow v\}$ (defined in Fig. 5), behaves like a standard substitution but on entering the body of an abstraction (second substitution rule for a lambda abstraction) it additionally demotes the value v. Demotion of v with respect to L is also needed when

$$
\begin{array}{lll}
(\texttt{let}^L\ y = s\ \texttt{in}\ e) & = \texttt{let}^L\ y = s\{x \Mapsto v\}\ \texttt{in} & \\
\quad \{x \Mapsto v\} & \quad (e\{x \Mapsto v^{\natural L}\}) & \\
y\{x \Mapsto v\} & = \begin{cases} y & \text{if } x \neq y \\ v & \text{if } x = y \end{cases} & \\
(\lambda(y,z).e)\{x \Mapsto v\} & = \begin{cases} \lambda(y,z).e & \text{if } x \in \{y,z\} \\ \lambda(y,z).(e\{x \Mapsto v^{\natural}\}) & \text{if } x \notin \{y,z\} \end{cases} & \\
\texttt{udf}\{x \Mapsto v\} & = \texttt{udf} & \\
(q\ell, i)\{x \Mapsto v\} & = (q\ell, i) & \\
(v_1(v_2))\{x \Mapsto v\} & = v_1\{x \Mapsto v\}(v_2\{x \Mapsto v\}) & \\
\texttt{new}^\ell\{x \Mapsto v\} & = \texttt{new}^\ell & \\
(v_1.a)\{x \Mapsto v\} & = v_1\{x \Mapsto v\}.a & \\
(v_1.a = v_2)\{x \Mapsto v\} & = v_1\{x \Mapsto v\}.a = (v_2\{x \Mapsto v\}) & \\
(v_1.a(v_2))\{x \Mapsto v\} & = v_1\{x \Mapsto v\}.a(v_2\{x \Mapsto v\}) & \\
(\triangledown^L e)\{x \Mapsto v\} & = \triangledown^L(e\{x \Mapsto v^{\natural L}\}) &
\end{array}
$$

Figure 5: Substitution.

applying the substitution to the body of a $\texttt{let}^L\ y = s\ \texttt{in}\ e$ expression and the mask expression $\triangledown^L e$. The L annotation on the expression declares the set of creation sites for which new objects may have been created during execution of e. All other cases are standard. Simultaneous substitution of x_i by v_i in e is defined in the obvious way and written as $e\{(x_1, \ldots, x_n) \Mapsto (v_1, \ldots, v_n)\}$.

4.4.4 Reduction Rules

Each read or write operation in the semantics requires the manipulation of two heaps H and H_0, as specified in the auxiliary read and write operations that operate directly on pairs of heaps.

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

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

Fig. 6 contains the reduction rules. The SMASK rule demotes the heaps with respect to a set L of locations. It moves all L-objects from the precise heap to the imprecise one. The SAPP rule is beta reduction with the extra proviso that the first argument is set to `udf`. The SLET rule is entirely standard. The SNEW rule creates a new, empty object at an address which is unused in both heaps and yields a precise reference to this object. The rules SREAD and SWRITE rely on the auxiliary read and write operations to dispatch the reference to the appropriate heap. The write operation returns `udf`. The method call SMCALL uses the same mechanism to access the method stored in the object and then performs beta reduction with the receiver object as for parameter. The SLET′ rule is a contextual rule analogous to S0LET′.

4.5 Properties of the Dynamic Semantics

This subsection formally states and proves some invariants for *reduction of elaborated expressions* in $\mathcal{RAC}$. **INV1** states that for each abstract location, there exists at most one object in the most-recent heap.

This invariant and the ones to follow do not hold for arbitrary configurations but only for configurations reachable from a closed, elaborated expression and empty heaps, *i.e.*, $\emptyset, \emptyset, e$.

Lemma 1 *Suppose that $\emptyset, \emptyset, e \longrightarrow^* H', H'_0, e'$ for a closed, elaborated expression e. Then **INV1** holds for H', H'_0, e'.*

$$
\begin{array}{rll}
\mathsf{Type} \ni & t & ::= \texttt{obj}(p) \mid (\Sigma, t \times t) \xrightarrow{L} (\Sigma, t) \mid \top \mid \texttt{udf} \\
& p & ::= \tilde{}L \mid @\ell \\
\mathsf{HeapType} \ni & r & \in \mathsf{Property} \xrightarrow{fin} \mathsf{Type} \\
\mathsf{GlobalEnv} \ni & \Omega & ::= \emptyset \mid \Omega(\ell : r) \\
\mathsf{LocalEnv} \ni & \Sigma & ::= \emptyset \mid \Sigma(\ell : r) \\
\mathsf{TypeEnv} \ni & \Gamma & ::= \emptyset \mid \Gamma(x : t)
\end{array}
$$

Figure 7: Type syntax.

Invariant **INV-LAM** for an expression e states that, for all lambda expressions $\lambda(y, x).e_0$ occurring in e, the body e_0 of the lambda never contains a precise reference $(@\ell, i)$.

Lemma 2 *Suppose that $\emptyset, \emptyset, e \longrightarrow^* H', H'_0, e'$. Then **INV-LAM** holds for H', H'_0, e'.*

Proof. The base case of the induction is the observation that the initial expression as entered by the programmer does not contain references.

For the inductive step, it must be shown: If **INV-LAM** holds for e and $H, H_0, e \longrightarrow H', H'_0, e'$, then **INV-LAM** holds for e'.

Simple inductive proof on the definition on $\longrightarrow$. Only substitution can insert a reference in the body of a lambda abstraction, but the definition of substitution demotes all such references before substituting into the body of the lambda. □

The invariant **INV-DIS** states that the domains of the summary heap and the most-recent heap do not overlap. Let H, H_0, e be a configuration.

INV-DIS $\mathrm{dom}(H) \cap \mathrm{dom}(H_0) = \emptyset$.

Lemma 3 *If $\emptyset, \emptyset, e \longrightarrow^* H', H'_0, e'$, then H', H'_0 fulfills **INV-DIS**.*

Proof. The base case is obvious as both heaps are empty.

For the inductive step, we have to prove: If H, H_0 fulfills **INV-DIS** and $H, H_0, e \longrightarrow H', H'_0, e'$, then H', H'_0 fulfills **INV-DIS**.

This proof is by induction on the definition of reduction $\longrightarrow$. Only the cases for mask and `new` expressions are interesting as the other reductions do not modify the heaps.

By SMASK, $H, H_0, \mathcal{E}[\triangledown^L e]$ reduces to $(H, H_0)^{\natural L}, \mathcal{E}[e^{\natural L}]$. That is $H' = H^{\natural L} \cup H_L$ and $H'_0 = H_0^{\natural L} \backslash H_L$ where $H_L = H_0^{\natural L} \downarrow \{(\ell, i) \mid \ell \in L, i \in \mathbf{N}\}$. Thus, the new configuration is $H', H'_0, \mathcal{E}[e^{\natural L}]$ and $\mathrm{dom}(H') \cap \mathrm{dom}(H'_0) = (\mathrm{dom}(H) \cup \mathrm{dom}(H_L)) \cap (\mathrm{dom}(H_0) \setminus \mathrm{dom}(H_L)) = \mathrm{dom}(H) \cap (\mathrm{dom}(H_0) \setminus \mathrm{dom}(H_L)) \cup \mathrm{dom}(H_L) \cap (\mathrm{dom}(H_0) \setminus \mathrm{dom}(H_L)) = \emptyset \cup \emptyset = \emptyset$.

By SNEW, $H, H_0, \mathcal{E}[\texttt{new}^\ell] \longrightarrow H, H_0\{(\ell, i) \mapsto \{\}\}, \mathcal{E}[(@\ell, i)]$, for some $(\ell, i) \notin \mathrm{dom}(H) \cup \mathrm{dom}(H_0)$. The last condition directly implies that $\mathrm{dom}(H) \cap (\mathrm{dom}(H_0) \cup \{(\ell, i)\}) = \emptyset$. □

4.6 Static Semantics

The type language defined in Figure 7 distinguishes object types, function types, the top type, and the type `udf` (undefined). An object type $\texttt{obj}(p)$ always refers indirectly to a heap type r either via a local environment Σ (corresponding to the most-recent heap) or via a global environment Ω (corresponding to the summary heap). Both environments are regarded in the obvious way as mappings from locations to heap types. The choice between the two environments is made by the reference type p. If $p = @\ell$, then p is a precise reference which refers to entry $\Sigma(\ell)$ in the local environment. If $p = \tilde{}L$, then p is an imprecise reference which refers to all the entries $\Omega(l)$, for $l \in L$, in the global environment. A heap

$$
\begin{array}{lllll}
\textsc{SMask} & H, H_0, \triangledown^L e & \longrightarrow & (H, H_0)^{\natural L}, e & \\
\textsc{SApp} & H, H_0, (\lambda(y,x).e)(v) & \longrightarrow & H, H_0, e\{(y,x) \mapsto (\texttt{udf}, v)\} & \\
\textsc{SLet} & H, H_0, \texttt{let}^L\ x = v\ \texttt{in}\ e & \longrightarrow & H, H_0, e\{x \mapsto v\} & \\
\textsc{SNew} & H, H_0, \texttt{new}^\ell & \longrightarrow & H, H_0\{(\ell,i) \mapsto \{\}\}, (@\ell, i) & \text{if } (\ell,i) \notin \mathrm{dom}(H) \cup \mathrm{dom}(H_0) \\
\textsc{SRead} & H, H_0, (q\ell, i).a & \longrightarrow & H, H_0, (H, H_0)(q\ell, i)(a) & \\
\textsc{SWrite} & H, H_0, (q\ell, i).a = v & \longrightarrow & (H, H_0)\{(q\ell, i)(a) \mapsto v\}, \texttt{udf} & \\
\textsc{SMCall} & H, H_0, (q\ell, i).a(v) & \longrightarrow & H, H_0, e\{(y,x) \mapsto ((q\ell, i), v)\} & \text{if } (H, H_0)(q\ell, i)(a) = \lambda(y,x).e
\end{array}
$$

$$
\textsc{SLet}'\ \frac{H, H_0, s \longrightarrow H', H_0', \mathcal{L}[v]}{H, H_0, \texttt{let}^L\ x = s\ \texttt{in}\ e'' \longrightarrow H', H_0', \mathcal{L}[\texttt{let}^L\ x = v\ \texttt{in}\ e'']}
$$

Figure 6: Refined small-step operational semantics.

type r is a record type describing heap contents. The type language includes recursive types by adopting a co-inductive interpretation for the grammar of types t. This convention avoids extending the type language with a special `rec` operator.

The function type $(\Sigma_2, t_0 \times t_2) \xrightarrow{L} (\Sigma_1, t_1)$ distinguishes two arguments, the self argument t_0 and the function parameter t_2, and a local environment Σ_2 considered as a third argument. A function yields a result type t_1 and an updated local environment Σ_1. Intuitively, the function threads the local environment through its body. The argument local environment describes the objects which are passed precisely to the function, whereas the resulting local environment describes those which are returned precisely from the function. The location set L on the arrow is the latent allocation effect of the function. It contains all locations for which the function expects no object in the most-recent heap on entry. For example, the function may allocate an object at such a location.

The top type $\top$ is the supertype of all other types. The type `udf` is a singleton type that only contains the value `udf`.

The typing environment Γ is standard.

There are two typing judgments,

- $\Omega, \Gamma \vdash_v v : t$ states the type of a value v and
- $\Omega, \Sigma, \Gamma \vdash_e e : t \Rightarrow L, \Sigma', \Gamma'$ states the type of an expression.

Besides the typing environment Γ, both typing judgments rely on the heap typings Ω for the summary heap. The type of an expression also depends on the most-recent heap Σ. As expressions perform computations that change the heaps, the expression judgment yields a new typing Σ' for the most-recent heap and a new type assumption Γ'. The new type assumption is needed because some object types may have to be downgraded from precise references to imprecise ones, *e.g.*, when a new object is created at the same location. The final outcome of the expression rule is a set L of locations. It stands for the allocation effect of e, *i.e.*, a superset of the set of locations at which an object may be created during execution of e.

4.6.1 Typing of Values

Subtyping is only needed for values in our system. Figure 8 defines the subtyping relation. It is reflexive, $\top$ is the maximal type, and an imprecise pointer type can be subsumed by one with a larger set of locations. A function type behaves invariantly in the arguments, covariantly in the result type, and also covariantly with respect to the effect annotation.

Lemma 4 *The subtyping relation $<:$ is transitive.*

Figure 9 contains the typing rules for values. The standard rules UNDEFINED, OBJECT, VARIABLE, and SUBSUMPTION present no surprises. The rule FUNCTION, however, is one of the main work horses of the type system. It has to deal with the complications illustrated by the examples in Section 3.2: precise references for location ℓ must not sneak past allocations of new ℓ-references.

This problem has two facets, both of which are treated using effects [14]. The first problem is that a variable may hold a value with a precise pointer type $\texttt{obj}(@\ell)$ when a function/method is invoked which allocates a new object at ℓ (see example (4) in Sec. 3.2). The solution is to equip each function type with an allocation effect, *i.e.*, the set L'' of locations for which the function may allocate a new object. If each call to the function is guarded by a $\triangledown^{L''}$, then all precise pointers $\ell \in L''$ are demoted before entering the function's body.

The second problem arises from the definition of substitution. It preventively demotes pointers as soon as the substitution proceeds into the body of a lambda abstraction. However, this demotion may be too conservative as the (demoted) imprecise pointer in the body

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

$$
\frac{\begin{array}{c}\mathrm{dom}(\Sigma_1) = \mathrm{dom}(\Sigma_1') \\ (\forall l \in \mathrm{dom}(\Sigma_1), \forall a \in \mathsf{Property})\ \Sigma_1(l)(a) <: \Sigma_1'(l)(a) \\ t_1 <: t_1' \qquad L \subseteq L'\end{array}}{(\Sigma_2, t_0 \times t_2) \xrightarrow{L} (\Sigma_1, t_1) <: (\Sigma_2, t_0 \times t_2) \xrightarrow{L'} (\Sigma_1', t_1')}
$$

Figure 8: Subtyping.

$$
\textsc{Undefined}\ \Omega, \Gamma \vdash_v \texttt{udf} : \texttt{udf} \qquad \textsc{Object}\ \Omega, \Gamma \vdash_v (q\ell, i) : \texttt{obj}(q\ell)
$$

$$
\textsc{Variable}\ \frac{\Gamma(x) = t}{\Omega, \Gamma \vdash_v x : t} \qquad \textsc{Subsumption}\ \frac{\Omega, \Gamma \vdash_v v : t \qquad t <: t'}{\Omega, \Gamma \vdash_v v : t'}
$$

$$
\textsc{Function}\ \frac{\begin{array}{c}\mathrm{dom}(\Sigma_1) \subseteq \mathrm{dom}(\Omega) \\ \mathrm{dom}(\Sigma_2) \subseteq \mathrm{dom}(\Omega) \qquad \mathrm{dom}(\Sigma_2) \cap L = \emptyset \\ L' \cup L'' \subseteq L \qquad \Gamma' = \Gamma \downarrow \mathrm{fv}(\lambda(y,x).e) \\ L' = \mathrm{Locs}(\Gamma') \qquad \Gamma'' = (\Gamma')^{\natural L'} \qquad \Omega, \Sigma_2 \vdash_\Gamma \Gamma' \lhd \Gamma'' \\ \Omega, \Sigma_2, \Gamma''(y : t_0)(x : t_2) \vdash_e e : t_1 \Rightarrow L'', \Sigma_1, \Gamma'''(y : t_0')(x : t_2')\end{array}}{\Omega, \Gamma \vdash_v \lambda(y,x).e : (\Sigma_2, t_0 \times t_2) \xrightarrow{L} (\Sigma_1, t_1)}
$$

Figure 9: Typing rules for values.

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

Figure 10: Locations in types and environments.

of the lambda may still be identical to the most recent precise pointer on invocation of the lambda, as in example (3) in Sec. 3.2.[7]

Again, effects and mask come to our rescue. Let Γ' be the typing environment of the lambda restricted to the free variables of the lambda. The set $L' = \mathrm{Locs}(\Gamma')$ in the function rule collects the precise and imprecise locations mentioned in the types of the free variables of the function (see Fig. 10 for the definition). It constructs the environment $\Gamma'' = (\Gamma')^{\natural L'}$ for the function body by demoting all types with respect to L', corresponding to the action of the substitution on pointers. It further adds L' to the effect of the function so that the mask operation around the function invocation moves any precise L'-pointer to the summary heap.

As every demotion implies a move of a reference from the most-recent heap to the summary heap, the corresponding heap typing environments have to be related at the same instant. The *flow relation* between the original and the demoted type environments serves exactly this purpose: $\Omega, \Sigma_2 \vdash_\Gamma \Gamma' \lhd \Gamma''$. It compares the entries in Γ' and Γ'' pointwise and make the summary environment Ω reflect the current state in Σ_2 wherever Γ' has a precise reference and Γ'' has an imprecise one. (See Fig. 12 for the definition of the flow relation.)

Finally, the rule uses the L-annotation on the function arrow to signal to the application site of the function the set of those locations that should be moved to the summary heap before entering the function. These comprise the precise references according to L' and the locations in L'' which indicate potential new object creations in the body of the lambda.

4.6.2 Typing of Expressions

Figure 11 contains the typing rules for expressions. The VALUE rule expresses that evaluation of a value has no effect on the heap. Hence, the environments are passed through unchanged.

The rule LET sequentializes computations. The L-effect of the header becomes the decoration of the `let` (which determines the way substitution works). The L-effects of header and body are merged to obtain the effect of the whole expression. The type

[7] A type system where precise pointers can be substituted in lambdas is conceivable, but it is not obvious how to proceed. Consider the following example:

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

After line two, the type environment would contain $x : \texttt{obj}(@\ell)$ and $f : ([\ell \mapsto [\,]], t_0 \times \texttt{obj}(@\ell)) \xrightarrow{L} ([\ell \mapsto [\,]], \texttt{obj}(@\ell) \times \texttt{obj}(@\ell))$, which is consistent with the substitution of a precise pointer for x in the body of f. However, after line three these types have to change to $x : \texttt{obj}(\tilde{}\ell)$ and $f : ([\ell \mapsto [\,]], t_0 \times \texttt{obj}(@\ell)) \xrightarrow{L} ([\ell \mapsto [\,]], \texttt{obj}(\tilde{}\ell) \times \texttt{obj}(@\ell))$. While the type change of x corresponds straightforwardly to an application of demotion, the change of f's type does not: it contains three occurrences of $\texttt{obj}(@\ell)$, but only one of them changes to $\tilde{}\ell$. Having read or write operations in the body of the function aggravates matters even further.
On the other hand, the operational semantics could be easily adapted to deal with this situation because at the time the type of f changes, there is only one precise pointer present in the body of the function.

$$
\textsc{Value}\quad \frac{\Omega, \Gamma \vdash_v v : t}{\Omega, \Sigma, \Gamma \vdash_e v : t \Rightarrow \Sigma, \Gamma}
$$

$$
\textsc{Let}\quad \frac{\begin{array}{c}\mathrm{dom}(\Sigma) \cap L_1 = \emptyset \qquad \Sigma = \Sigma^{\natural L1} \qquad \Gamma = \Gamma^{\natural L1} \\ L_1 \subseteq L \qquad L_2 \subseteq L \qquad \Omega, \Sigma, \Gamma \vdash_e s_1 : t_1 \Rightarrow L_1, \Sigma_1, \Gamma_1 \\ \Omega, \Sigma_1, \Gamma_1(x : t_1) \vdash_e e_2 : t_2 \Rightarrow L_2, \Sigma_2, \Gamma_2(x : t_1')\end{array}}{\Omega, \Sigma, \Gamma \vdash_e \texttt{let}^{L_1}\ x = s_1 \texttt{ in } e_2 : t_2 \Rightarrow L, \Sigma_2, \Gamma_2}
$$

$$
\textsc{Mask}\quad \frac{\begin{array}{c}(\forall l \in L)\ \Omega, \Sigma \vdash_t \texttt{obj}(@l) \lhd \texttt{obj}(\tilde{}l) \\ \Sigma' = \Sigma^{\natural L} \uparrow L \qquad \Omega, \Sigma', \Gamma^{\natural L} \vdash_e e : t \Rightarrow L, \Sigma'', \Gamma''\end{array}}{\Omega, \Sigma, \Gamma \vdash_e \nabla^L e : t \Rightarrow L, \Sigma'', \Gamma''}
$$

$$
\textsc{Function Call}\quad \frac{\begin{array}{c}\Gamma = \Gamma^{\natural L} \\ \Sigma = \Sigma^{\natural L} \qquad \mathrm{dom}(\Sigma) \cap L = \emptyset \qquad \Omega, \Sigma \vdash_t \texttt{udf} \lhd t_0 \\ \Omega, \Gamma \vdash_v v_2 : t_2 \qquad \Omega, \Gamma \vdash_v v_1 : (\Sigma, t_0 \times t_2) \xrightarrow{L} (\Sigma_1, t_1)\end{array}}{\Omega, \Sigma, \Gamma \vdash_e v_1(v_2) : t_1 \Rightarrow L, \Sigma_1, \Gamma}
$$

$$
\textsc{Method Call}\quad \frac{\begin{array}{c}\Gamma = \Gamma^{\natural L} \qquad \Sigma = \Sigma^{\natural L} \\ \mathrm{dom}(\Sigma) \cap L = \emptyset \qquad \Omega, \Gamma \vdash_v v_1 : \texttt{obj}(p) \qquad \texttt{obj}(p) <: t_0 \\ \Omega, \Gamma \vdash_v v_2 : t_2 \qquad \Omega \vdash_r p.a : (\Sigma, t_0 \times t_2) \xrightarrow{L} (\Sigma_1, t_1)\end{array}}{\Omega, \Sigma, \Gamma \vdash_e v_1.a(v_2) : t_1 \Rightarrow L, \Sigma_1, \Gamma}
$$

$$
\textsc{New}\quad \frac{\Gamma = \Gamma^{\natural \ell} \qquad \ell \in \mathrm{dom}(\Omega) \qquad l \notin \mathrm{dom}(\Sigma) \qquad \Sigma = \Sigma^{\natural \ell}}{\Omega, \Sigma, \Gamma \vdash_e \texttt{new}^\ell : \texttt{obj}(@\ell) \Rightarrow \{\ell\}, \Sigma(\ell \mapsto \{\}), \Gamma}
$$

$$
\textsc{Read}\quad \frac{\Omega, \Gamma \vdash_v v : \texttt{obj}(p) \qquad \Omega, \Sigma \vdash_r p.a : t}{\Omega, \Sigma, \Gamma \vdash_e v.a : t \Rightarrow \emptyset, \Sigma, \Gamma}
$$

$$
\textsc{Write}\quad \frac{\begin{array}{c}\Omega, \Gamma \vdash_v v : \texttt{obj}(ql) \\ \Omega, \Gamma \vdash_v v' : t' \qquad \Omega, \Sigma \vdash_w ql.a = t' \Rightarrow \Sigma'\end{array}}{\Omega, \Sigma, \Gamma \vdash_e v.a = v' : \texttt{udf} \Rightarrow \emptyset, \Sigma', \Gamma}
$$

Figure 11: Typing rules for expressions.

and most-recent environments are threaded through the header to the body. The L_1-effect of the header also determines the set of locations that must **not** be passed precisely into the `let` header: The conditions $\mathrm{dom}(\Sigma) \cap L_1 = \emptyset$, $\Sigma = \Sigma^{\natural L_1}$, and $\Gamma = \Gamma^{\natural L_1}$ enforce this restriction, which is crucial in the proof of progress.

The MASK rule types the expression that moves all precise references with locations in L from the most-recent heap to the summary heap. As in the FUNCTION rule, this move must be reflected in the summary heap environment with flow judgments: $(\forall l \in L)\ \Omega, \Sigma \vdash_t \texttt{obj}(@l) \lhd \texttt{obj}(\tilde{}l)$. In addition, these references are removed from the most-recent heap: $\Sigma' = \Sigma^{\natural L} \uparrow L$. The most-recent heap and the demoted type environment are threaded through the body of the mask expression. The result type is taken over. The final Σ'' may not have any relation with the incoming Σ'.

The FUNCTION CALL rule is driven by the L-effect of the function type. It requires that Γ and Σ do not contain L-precise references (by using a $\nabla^L \ldots$ expression). It sets the self parameter of the function to `udf`, matches the second argument type with the type of the argument v_2, and yields the return type t_1.

The METHOD CALL rule first extracts the function's type from the property of the object and then works similarly. The type of the first argument is the type of the receiver object. The preconditions can be fulfilled by wrapping the method call in a mask expression.

The NEW rule defines the typing for object creation. As objects are always created exactly and without preinitialized properties, the return type is $\texttt{obj}(@\ell)$ and the most-recent heap environment registers the binding $\ell \mapsto \{\}$. The other preconditions guarantee that no precise ℓ-reference exists when the new ℓ-object is created.

The READ rule is straightforward. It relies on an auxiliary judgment that performs a lookup either in the summary environment or in the most-recent one, depending on the precision of the reference.

The WRITE rule is similar. The auxiliary judgment returns a new most-recent environment because it performs a strong update on precise reference types. The write expression returns `udf`.

4.6.3 Auxiliary Judgments

Figure 12 defines some auxiliary judgments used in the typing rules. The first group of rules defines the flow judgment $\Omega, \Sigma \vdash_t t \lhd t'$. It forces the type of a reference to flow from the most-recent environment Σ to the summary environment Ω if that reference has been converted from precise to imprecise (from t to t'). The first rule, dealing with the object type, is the most important one: If an ℓ-reference changes from precise to imprecise, then the corresponding entries in Σ and Ω must be flow related. The last rule then pushes the flow into the objects' properties, and then (most of the time) the second rule relates the property types by subtyping. For example, if $\Sigma(\ell) = \texttt{udf}$ then $\Omega(\ell)$ can be either `udf` or $\top$.

The second group defines the read judgment $\Omega, \Sigma \vdash_r p.a : t$, which executes the abstract read operation from property a of reference p. The type of a precise reference comes straight from the most-recent environment. The type of an imprecise reference spread over locations in L is a supertype of the types at all locations.

The third group defines the write judgment $\Omega, \Sigma \vdash_w p.a = t \Rightarrow \Sigma'$, 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 most-recent environment Σ' in case p is a precise reference. Writing to an imprecise reference (first rule) just places a constraint on the summary environment: the type in the store must subsume the type t of the written value. Writing to a precise reference does not affect the summary environment, but changes the respective location in the most-recent environment.

4.7 Prototype Extension

This subsection demonstrates how to extend $\mathcal{RAC}$ with the full 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 `pt` of the new object to v. This property is not to be used in user code.

The read reduction rule SREAD$'$ replaces the SREAD rule in Fig. 6. The read operation first looks into the value v obtained by reading the object itself. It returns v unless $v = \texttt{udf}$. Otherwise, if a prototype is defined for the object, it delegates the lookup to the prototype. If the property is undefined and the prototype is not an object, the read operation returns `udf`.

The typing rule for `new` is revised to deal with the prototype argument and to install it 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 object along the prototype chain, returning the value when the property is found and passing the property read on to the prototype if one exists.

$$\boxed{\Omega, \Sigma \vdash_t t \lhd t'}$$

$$\frac{\ell \in L \quad \ell \notin \mathrm{dom}(\Sigma) \vee \Omega, \Sigma \vdash_h \Sigma(\ell) \lhd \Omega(\ell)}{\Omega, \Sigma \vdash_t \texttt{obj}(@\ell) \lhd \texttt{obj}(\tilde{}L)}$$

$$\frac{t <: t'}{\Omega, \Sigma \vdash_t t \lhd t'} \qquad \frac{(\forall a \in \mathsf{Property})\ \Omega, \Sigma \vdash_t r(a) \lhd r'(a)}{\Omega, \Sigma \vdash_h r \lhd r'}$$

$$\boxed{\Omega, \Sigma \vdash_r p.a : t} \qquad \frac{\Sigma(\ell)(a) = t}{\Omega, \Sigma \vdash_r @\ell.a : t}$$

$$\frac{(\forall \ell \in L)\ \Omega, \Sigma \vdash_r \tilde{}\ell.a : t}{\Omega, \Sigma \vdash_r \tilde{}L.a : t} \qquad \frac{\Omega(\ell)(a) = t}{\Omega, \Sigma \vdash_r \tilde{}\ell.a : t}$$

$$\boxed{\Omega, \Sigma \vdash_w p.a = t \Rightarrow \Sigma'} \qquad \frac{(\forall \ell \in L)\ t <: \Omega(\ell)(a)}{\Omega, \Sigma \vdash_w \tilde{}L.a = t \Rightarrow \Sigma}$$

$$\Omega, \Sigma \vdash_w @\ell.a = t \Rightarrow \Sigma(\ell : \Sigma(\ell)[a \mapsto t])$$

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

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].

5. Metatheory

This section presents the type soundness result for $\mathcal{RAC}$. The basic structure of the proof follows Felleisen and Wright [30], with 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.

A few auxiliary lemmas are needed to prove the substitution lemma: the subtype relation is a subset of the flow relation (Lemma 5), the influence of the effect on the typing environment (Lemma 6), and the typing of a value after demotion (Lemma 7).

Lemma 5 *If $t <: t'$ holds, then $\forall \Omega, \Sigma : \Omega, \Sigma \vdash_t t \lhd t'$*

Lemma 6 *If $\Omega, \Sigma, \Gamma \vdash_e e : t \Rightarrow L, \Sigma', \Gamma'$ then $\Gamma' = \Gamma^{\natural L}$.*

Lemma 6 says that the typing environment that is returned from the expression judgment can be computed from the incoming typing environment by applying demotion to it. In consequence, the judgment could be simplified by omitting the returned typing environment. However, this choice diminishes the readability of the rules for `let` and `mask` expressions and thus leads to a less intuitive formulation of the type system.

Lemma 7 *If $\Omega, \Sigma, \Gamma \vdash_v v : t$, then $\Omega, \Sigma', \Gamma^{\natural L} \vdash_v v^{\natural L} : t^{\natural L}$, for arbitrary Σ'.*

Proof. Observe that $t <: t'$ implies $t^{\natural L} <: t'^{\natural L}$. □

Lemma 8 (Substitution) *Suppose that $\Omega, \Sigma, \emptyset \vdash_v v : t_x$ and $\Omega, \Sigma, \Gamma(x : t_x) \vdash_e e : t_0 \Rightarrow L, \Sigma', \Gamma'$.*
Then $\Omega, \Sigma, \Gamma \vdash_e e\{x \Mapsto v\} : t_0 \Rightarrow L, \Sigma', \Gamma'$.

To state and prove typing preservation and then progress, we extend the notion of typing to configurations with closed expressions in the obvious way as shown in Fig. 14.

Additional syntax

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

Additional reductions

$$\begin{array}{lllll} \textsc{New}' & H, H_0, & \longrightarrow & H, H_0[(\ell, j) \mapsto \{\texttt{pt} \mapsto v\}], & \\ & \texttt{new}^{\ell}(v) & & (@\ell, j) & \\ & & & \text{if } (\ell, j) \notin \text{dom}(H) \cup \text{dom}(H_0) & \\ \textsc{Read}' & H, H_0, & \longrightarrow & H, H_0, & \\ & (p, i).a & & \mathit{read}(H, H_0, (p, i), a) & \end{array}$$

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

where $pt = (H, H_0)(p, i)(\texttt{pt})$
$v = (H, H_0)(p, i)(a)$

Changes in the static semantics

New'

$$\frac{\Gamma = \Gamma^{\natural\ell} \qquad \ell \in \text{dom}(\Omega) \setminus \text{dom}(\Sigma) \qquad \Sigma = \Sigma^{\natural\ell} \qquad \Omega, \Sigma, \Gamma \vdash_v v : t}{\Omega, \Sigma, \Gamma \vdash_e \texttt{new}^{\ell}(v) : \texttt{obj}(@\ell) \Rightarrow \{\ell\}, \Sigma(\ell \mapsto \{\texttt{pt} \mapsto t\}), \Gamma}$$

$$\frac{(\forall \ell \in L)\ \Omega, \Sigma \vdash_r \tilde{}\ell.a : t \qquad t <: t'}{\Omega, \Sigma \vdash_r \tilde{}L.a : t'}$$

$$\frac{(\Omega, \Sigma)(p)(a) = t \neq \texttt{udf}}{\Omega, \Sigma \vdash_r p.a : t}$$

$$\frac{(\Omega, \Sigma)(p)(a) = \texttt{udf} \qquad (\Omega, \Sigma)(p)(\texttt{pt}) = \texttt{obj}(q) \qquad \Omega, \Sigma \vdash_r q.a : t}{\Omega, \Sigma \vdash_r p.a : t}$$

$$\frac{(\Omega, \Sigma)(p)(a) = \texttt{udf} \qquad (\Omega, \Sigma)(p)(\texttt{pt}) \neq \texttt{obj}(())}{\Omega, \Sigma \vdash_r p.a : \texttt{udf}}$$

Figure 13: Extension to support prototypes.

$$\frac{\Omega, \Sigma \Vdash H, H_0 \qquad \Omega, \Sigma, \emptyset \vdash_e e : t \Rightarrow L, \Sigma', \emptyset}{\Omega, \Sigma \Vdash_e H, H_0, e : t \Rightarrow L, \Sigma'}$$

$$\frac{\begin{array}{c} \text{dom}(\Sigma) \subseteq \text{dom}(\Omega) \\ (\forall (\ell, i) \in \text{dom}(H_0))\ \ell \in \text{dom}(\Sigma) \wedge \Omega, \Sigma \Vdash_o H_0(\ell, i) : \Sigma(\ell)) \\ (\forall (\ell, i) \in \text{dom}(H))\ \ell \in \text{dom}(\Omega) \wedge \Omega, \Sigma \Vdash_o H(\ell, i) : \Omega(\ell)) \end{array}}{\Omega, \Sigma \Vdash H, H_0}$$

$$\frac{(\forall a \in \text{dom}(h))\ a \in \text{dom}(r) \wedge \Omega, \Sigma, \emptyset \vdash_v h(a) : r(a)}{\Omega, \Sigma \Vdash_o h : r}$$

Figure 14: Typing of heaps and configurations.

Lemma 9 (Typing Preservation) *Suppose that* $\Omega, \Sigma \Vdash_e H, H_0, e : t \Rightarrow L', \Sigma''$ *and* $H, H_0, e \longrightarrow H', H_0', e'$.

Then there exists some Σ', t', *and* L'' *such that* $\Omega, \Sigma' \Vdash_e H', H_0', e' : t' \Rightarrow L'', \Sigma''$ *with* $\Omega, \Sigma' \vdash_t t' \lhd t$ *and* $L'' \subseteq L'$.

Working towards progress, the invariant **INV-CLS** states closedness of a configuration with respect to the heap: Any reference contained in one of the heaps or in the expression is defined in one of the heaps. An auxiliary definition simplifies the statement of the invariant. Define $(q\ell, i) \in H, H_0$ by

- $(\tilde{}\ell, i) \in H, H_0$ iff $(\ell, i) \in \text{dom}(H) \cup \text{dom}(H_0)$ and
- $(@\ell, i) \in H, H_0$ iff $(\ell, i) \in \text{dom}(H_0)$.

While this relation leads to a provable invariant, it is insufficient for proving progress: Additional information is needed that states when an imprecise reference can definitely be found in the summary heap! Such references appear exactly in *unblocked contexts*:

An unblocked context $\mathcal{U}^{\ell}$ for an imprecise reference $(\tilde{}\ell, i)$ is defined for all L such that $\ell \notin L$ and arbitrary L' as

$$\begin{array}{lll} \mathcal{U}^{\ell} & ::= & \Box \mid v(\Box) \mid \Box.a(v) \mid v.a(\Box) \mid \nabla^{L}\mathcal{U}^{\ell} \mid \Box.a \mid \Box.a = v \\ & \mid & v.a = \Box \mid \texttt{let}^{L'}\ x = \mathcal{U}^{\ell}\ \texttt{in}\ e \mid \texttt{let}^{L}\ x = s\ \texttt{in}\ \mathcal{U}^{\ell} \\ & \mid & \lambda^{L}(y, x).\Box \end{array}$$

where the L-annotation on the lambda expression stands for the effect of the lambda's body.

INV-CLS holds for a configuration H, H_0, e if:

1. e is closed.
2. If $(q\ell, i)$ occurs in e, then $(q\ell, i) \in H, H_0$.
3. If $e = \mathcal{U}^{\ell}[(\tilde{}\ell, i)]$, then $(\ell, i) \in \text{dom}(H)$.
4. For all $(\ell, i) \in \text{dom}(H) \cup \text{dom}(H_0)$, if $h = (H \cup H_0)(\ell, i)$ then, for all $a \in \text{dom}(h)$,
   (a) if $(q\ell', i')$ occurs in $h(a)$, then $(q\ell', i') \in H, H_0$ and
   (b) if $h(a) = (\tilde{}\ell', i')$, then $(\ell', i') \in \text{dom}(H)$.

The invariant **INV-CLS** holds only for typed configurations.

Lemma 10 *Suppose that* $\Omega, \Sigma \Vdash_e H, H_0, e : t \Rightarrow L, \Sigma'$.

If H, H_0, e *fulfills* ***INV-CLS*** *and* $H, H_0, e \longrightarrow H', H_0', e'$ *then* H', H_0', e' *fulfills* ***INV-CLS***.

Proof. By induction on the definition of reduction $\longrightarrow$.

Lemma 11 (Progress) *Suppose that* $\Omega, \Sigma \Vdash_e H, H_0, e : t \Rightarrow L, \Sigma'$, *so that* ***INV-CLS*** *holds for the configuration.*

Then either e *is a value or there exists* H', H_0', e' *such that* $H, H_0, e \longrightarrow H', H_0', e'$.

Proof. Induction on e. Some illustrative cases are:

- Case $e = (\tilde{}l, i).a$. **INV-CLS** yields that $(l, i) \in \text{dom}(H)$. Hence SRead is applicable.
- Case $e = (@l, i).a$. Analogous to $e = (\tilde{}l, i).a$.
- Case $e = (\tilde{}l, i).a(v)$. The invariant **INV-CLS** yields that $(l, i) \in \text{dom}(H)$. Inversion of the typing and heap consistency ensures that $H(l, i)(a)$ is a function. Hence SMCall is applicable.

Type soundness can now be concluded in the usual way from type preservation and progress.

6. Extensions

The recency-aware type system requires some extensions to become usable in practice. For the sake of simplicity, the paper does not include their formalization. However, they are straightforward to add and do not lead to new insights.

Recursion can be handled in the static semantics by a simple change of the typing rule FUNCTION to deal with recursive functions $\texttt{fix}f(y,z).e$. The change is to add the assumption $(f : (\Sigma_2, t_0 \times t_2) \xrightarrow{L} (\Sigma_1, t_1))$ to Γ'' when deriving the type of e. The change in the operational semantics is also straightforward:

$$\begin{array}{rl} & H, H_0, (\texttt{fix}f(y,x).e)(v) \\ \longrightarrow & H, H_0, e\{(f,y,x) \Mapsto (\texttt{fix}f(y,x).e, \texttt{udf}, v)\} \end{array}$$

In JavaScript, there is a subtle difference between an object that lacks a property and an object that has a property set to `udf`. While this difference cannot be observed by dereferencing the property, a prototype object can be used to distinguish these two cases. To address this phenomenon, our type system would have to be extended with another `unset` state for the properties of an object. This `unset` state would be propagated by the type system just like any other type of a record field.

The top type $\top$ used in the type system is a crude approximation of the behavior desired in practice. As the type system does not provide elimination rules for the $\top$ type, a value of this type cannot be processed any more. Instead of the uninformative $\top$ we would rather include a dynamic type or a discriminative sum type that subsumes the types of the values that flow into it. Again, we have avoided to model such a dynamic type because we have already investigated this issue in previous work [27] and its addition to $\mathcal{RAC}$ would obscure the current presentation.

7. Related Work

The original ideas of using creation sites for abstracting heap structures as well as the per-program-point approximation of the heap are due to Jones and Muchnick [17, 18].

The notion of strong update is due to Chase et al [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 information in the (per-program-point) most-recent environments and it comes with a correctness proof.

Our type system has some relation 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 creation site. Thus, answering a question raised in previous work [16], the idea of their approach can be extended to a higher-order language.

Smith, Walker, and Morrisett [25, 29] have considered alias types as an extension of linear types for typing low-level intermediate languages. Their calculus models object initialization, *i.e.*, type changing assignments to heap records, thus it would be a suitable target language for modeling type change in a scripting language. Alias types rely on separating the types of pointer values from the actual store contents, just like our division between the object types and the heap typing environments. Their pointer values are singleton types of the form $\texttt{ptr}(l)$, where l stands for a single store location, they rely on existential quantification to specify recursive data structures, and explicit coercion operations are needed to pack and unpack existentials and to roll and unroll recursive data types. In contrast, our object type $\texttt{obj}(@\ell)$ with a precise reference is also a singleton type standing for a particular location but this location depends on the current execution state. Our $\texttt{obj}(\tilde{L})$ type has an existential-type flavor and it can model recursive data structures. However, the precision is significantly lower than with alias types because our setting does not support operations to unpack and to unroll. This difference is not surprising as the alias types system is a prescriptive, explicitly typed calculus with decidable type checking whereas our calculus is descriptive, implicitly typed, and has a typing algorithm.

Fähndrich and Xia's delayed types [13] 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 typechecking 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 information about an object exactly as long as possible and reverts to less precise summary information when it is no longer avoidable. 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 [19] 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. The object loses its special status only when the next object is allocated at the same abstract location. Objects that are not most-recent can also be updated, but the effect of this update cannot be traced precisely in the type.

Previous work involving the second author [16] has defined 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 [21] 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 Foster et al's "Flow-sensitive type qualifiers" [12] has properties very similar to abstract counting.

Anderson et al [2] define a type system for JavaScript that comes with a type inference algorithm. Their work considers a similar core language than our work. Their type system is based on an extension of record types by a definedness indicator on each record component. The latter records whether a record component is definitively initialized or whether it may be uninitialized. Thus, they do not model type change and they do not consider prototypes. We conjecture that our system can type check all programs that Anderson's system can check (it can type check all examples in that work). In addition, our system treats type change and prototypes.

In previous work, the second author has proposed a type system for JavaScript [27]. 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 fully complementary to the previous work. A practical system would have to build on a combination

of both works and it would have to encompass the entirety of the language as formalized by Maffeis and others [20].

While the two previously discussed works may be called descriptive, there are also researchers treating scripting language typing from the prescriptive point of view. Prominent examples are the work on typed Scheme [28] and the work on the next version of JavaScript [9]. Their goal is to enrich untyped languages gradually with type assertions and contracts and thus transition over time to an explicitly typed language. The work on gradual typing [24] has similar goals.

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 the said extension and encapsulation. Their calculus distinguishes prototypes (which are extensible and overridable records, but do not allow for subtyping) from objects (which are no longer modifiable, but admit 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 [10], which considers a functional calculus with similar features and with a mytype mechanism which does not include a typing algorithm.

8. Conclusion

The idea of a recency abstraction can be fruitfully transported to a type system. This type system is amenable to analyzing programs in scripting languages, in particular handling object initialization. Its distinguishing feature compared to previous approaches is the ability to handle type changes and prototypes precisely.

An implementation of a type inference algorithm is work in progress. Once that implementation is in place, an evaluation of its usefulness on fragments of JavaScript programs can take place.

As future work, location and store polymorphism may be worthwhile additions. In the present system, a function can only manipulate values created at a fixed set of locations and the typing of the ignored part of the most-recent heap must coincide at all call sites of the function.

Another worthwhile avenue would be to consider adding type states [26] to this kind of type system. However, it is unclear if the present facilities are sufficient.

Acknowledgments

Thanks to the anonymous reviewers for their extensive, helpful comments.

References

[1] Rita Z. Altucher and William Landi. An extended form of must alias analysis for dynamic allocation. In *Proc. 1995 ACM Symp. POPL*, pages 74–84, San Francisco, CA, USA, January 1995. ACM Press.

[2] Christopher Anderson, Paola Giannini, and Sophia Drossopoulou. Towards type inference for JavaScript. In *19th ECOOP*, number 3586 in LNCS, Glasgow, Scotland, July 2005. Springer.

[3] Gogul Balakrishnan and Thomas W. Reps. Recency-abstraction for heap-allocated storage. In Kwangkeun Yi, editor, *SAS*, number 4134, pages 221–239. Springer, 2006.

[4] Viviana Bono and Kathleen Fisher. An imperative, first-order calculus with object extension. In Eric Jul, editor, *12th ECOOP*, number 1445 in LNCS, pages 462–497, Brussels, Belgium, July 1998. Springer.

[5] John Boyland, James Noble, and William Retert. Capabilities for sharing: A generalisation of uniqueness and read-only. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 2–27, London, UK, 2001. Springer-Verlag.

[6] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proc. PLDI '90*, pages 296–310, White Plains, New York, USA, June 1990. ACM.

[7] dojo the javascript toolkit. `http://dojotoolkit.org/`, 2008.

[8] ECMAScript Language Specification. `http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf`, December 1999. ECMA International, ECMA-262, 3rd edition.

[9] ECMAScript, the language of the web. `http://www.ecmascript.org/`, 2008.

[10] Kathleen Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Standford University, 1996. Stanford Computer Science Technical Report STAN-CS-TR-98-1602.

[11] Cormac Flanagan, Amr Sabry, Bruce F. Duba, and Matthias Felleisen. The essence of compiling with continuations. In *Proc. 1993 PLDI*, pages 237–247, Albuquerque, NM, USA, June 1993.

[12] Jeffrey S. Foster, Tachio Terauchi, and Alex Aiken. Flow-sensitive type qualifiers. In PLDI 2002 [22], pages 1–12.

[13] Manuel Fähndrich and Songtao Xia. Establishing object invariants with delayed types. In *Proc. 22nd ACM Conf. OOPSLA*, pages 337–350, Montreal, QC, CA, 2007. ACM Press, New York.

[14] David Gifford and John Lucassen. Integrating functional and imperative programming. In *Proc. 1986 ACM Conference on Lisp and Functional Programming*, pages 28–38, 1986.

[15] Google web toolkit. `http://code.google.com/webtoolkit/`, 2008.

[16] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: Must-alias analysis for higher-order languages. In Luca Cardelli, editor, *Proc. 25th ACM Symp. POPL*, pages 329–341, San Diego, CA, USA, January 1998. ACM Press.

[17] Neil D. Jones and Stephen Muchnick. Flow analysis and optimization of Lisp-like languages. In *Proc. of the 6th ACM Symp. POPL*, pages 244–256. ACM Press, 1979.

[18] Neil D. Jones and Steven S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *POPL '82: Proceedings of the 9th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 66–74, 1982.

[19] Matthew Kehrt and Jonathan Aldrich. A theory of linear objects. In *FOOL 2008*, San Francisco, CA, USA, January 2008. `http://fool08.kuis.kyoto-u.ac.jp/kehrt.pdf`.

[20] Sergio Maffeis, John C. Mitchell, and Ankur Taly. An operational semantics for JavaScript. In G. Ramalingam, editor, *APLAS 2008*, number 5356 in LNCS, pages 307–325, Bangalore, India, December 2008. Springer.

[21] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In Julia Lawall, editor, *Proc. ICFP 2006*, pages 13–25, Portland, Oregon, USA, September 2006. ACM Press, New York.

[22] *Proc. 2002 PLDI*, Berlin, Germany, June 2002. ACM Press.

[23] Scriptaculous. `http://script.aculo.us/`, 2008.

[24] Jeremy Siek and Walid Taha. Gradual typing for objects. In Erik Ernst, editor, *21st ECOOP*, volume 4609 of *LNCS*, pages 2–27, Berlin, Germany, July 2007. Springer.

[25] Frederick Smith, David Walker, and J. Gregory Morrisett. Alias types. In Gert Smolka, editor, *Proc. 9th ESOP*, number 1782 in LNCS, pages 366–381, Berlin, Germany, March 2000. Springer.

[26] Robert E. Strom and Shaula Yemini. Typestate: A programming language concept for enhancing software reliability. *IEEE Trans. Softw. Eng.*, 12(1):157–171, 1986.

[27] Peter Thiemann. Towards a type system for analyzing JavaScript programs. In *Proc. 14th ESOP*, number 3444 in LNCS, pages 408–422, Edinburgh, Scotland, April 2005. Springer.

[28] Sam Tobin-Hochstadt and Matthias Felleisen. The design and

implementation of typed scheme. In Phil Wadler, editor, *Proc. 35th ACM Symp. POPL*, pages 395–406, San Francisco, CA, USA, January 2008. ACM Press.

[29] David Walker and Greg Morrisett. Alias types for recursive data structures. In Robert Harper, editor, *Proc. ACM Workshop Types in Compilation (TIC'00)*, number 2071 in LNCS, pages 177–206, Montréal, Canada, September 2000. Springer.

[30] Andrew Wright and Matthias Felleisen. A syntactic approach to type soundness. *Information and Computation*, 115(1):38–94, 1994.