Type Checking for JavaScript

Christopher Anderson [1,3]

Department of Computing, Imperial College of Science, Technology and Medicine 180 Queen's Gate, London SW7 2BZ, U.K.

Paola Giannini [1,2,4]

Dipartimento di Informatica, Università del Piemonte Orientale, Spalto Marengo 33, Alessandria, Italy.

Abstract

JavaScript is a powerful imperative object based language made popular by its use in web pages. It supports flexible program development by allowing dynamic addition of members to objects. Code is dynamically typed: a runtime access to a non-existing member causes an error.

We suggest a static type system for JavaScript that will detect such type errors. Therefore, programmers can benefit from the flexible programming style offered by JavaScript and from the safety offered by a static type system.

We demonstrate our type system with a formalism of JavaScript, #1. Our types are structural. Members of #1 type are classified into *definite* and *potential*. A potential member becomes definite upon assignment. We outline a proof that our type system is sound.

1 Introduction

JavaScript (see [12]) is a powerful imperative object based language made popular by its use in web pages. JavaScript supports flexible program development by allowing dynamic addition of members to objects.

JavaScript code is embedded directly in web pages and interpreted as the page is loaded. Code is dynamically typed and if at runtime a field is accessed

[1] Work partly supported by DART, European Commission Research Directorates, IST-01-6-1A

[2] MURST Cofin'02 project McTati

[3] Email: `cla97@doc.ic.ac.uk`

[4] Email: `giannini@mfn.unipmn.it`

or method called that does not exist then a runtime type error is generated. When such errors occur the user is usually presented with an error message.

We suggest a static type system for JavaScript that will detect type errors that are currently only detected at runtime. Therefore, programmers can benefit from the flexible programming style offered by JavaScript and from the safety offered by a static type system. We demonstrate our type systems with a formalism of JavaScript, JS_0. JS_0 supports the standard JavaScript flexible features, e.g. functions creating objects, and dynamic addition/reassignment of fields and methods.

Our type systems tackles the following challenges introduced by the flexible features of JS_0:

- JS_0 object structure is determined by assignment of members,
- JS_0 objects can have members added to them after they have been created,
- JS_0 methods are created by assigning functions to members,
- JS_0 methods can be shared among objects, and
- JS_0 functions can have three different roles: creating objects, methods of objects and global functions.

We address these issues with an explicitly typed version of JS_0, JS_0^{T}. JS_0^{T} uses structural types of the form: $\mathsf{t} = \mu\ \alpha.\ \ll \mathsf{m}_1 : \mathsf{t}_1 \cdots \mathsf{m}_n : \mathsf{t}_n \gg$ #1

#1

This paper is organized as follows. In Section 2 we present an example introducing the features of JS_0 and JS_0^{T}. In Section 3 we define the syntax of JS_0 and its operational semantics, and in Section 4 we give a typed version of JS_0, JS_0^{T}. The proof of soundness JS_0^{T} is outlined in Section 5. Finally, in Section 6 we compare our work with others and outline our future directions.

2 Example

We start with an example demonstrating the classic untyped style of programming seen in JavaScript. In Figure 1 we give an example that describes a scenario with people and their jobs. We define functions `Person`, `moneyTrans`, `employPerson`. The code preceded by the comment `//Main` is the entry point into the program.

Figure 1 demonstrates:

- creating objects using functions (line 16 and 17),
- implicit creation of members in objects through assignment (lines 2 and 3),
- acquiring methods through assignment of a function to a member (line 3),
- method call, with `paul.payMe(10)` binding `paul` to `this` when `moneyTrans` is executed (line 18),
- addition of members after object creation, with `employPerson` adding member `boss` (line 12), and

- global function call, with `employPerson(paul,john)` being called without a receiver (line 18).

We now look at the same example in the context of a typed version of JavaScript. In the example we use a syntax slightly more liberal from the syntax of $\mathrm{JS}_0^{\mathsf{T}}$, allowing functions with more than one parameter and variable declarations. Figure 2 gives a typed version of Figure 1 where:

- MT is $(\ll \texttt{money} : \mathsf{Int} \gg, \mathsf{Int}) \rightarrow \mathsf{Int}$, and
- T is $\mu\ \alpha.\ \ll \texttt{money} : \mathsf{Int}, \texttt{payMe} : \mathrm{MT}, \texttt{boss} : \alpha? \gg$.

The first thing to note is that unlike JavaScript for functions there are type annotations for the formal parameters, (lines 1, 8, 13, and 14), the return type (lines 1, 8, and 15), and the type of the metavariable `this` which the function may be method of (lines 2, 9, and 17). Moreover, we declared the type of the two variables `paul` and `john` (lines 22, and 23).

Secondly, the *object types* are structural, comprising a list of members each with their own type. Consider the return type of function `Person` on line 1:

$$\ll \texttt{money} : \mathsf{Int}, \texttt{payMe} : \mathrm{MT}, \texttt{boss} : \mathrm{T}? \gg \tag{1}$$

Type (1) is the type of an object which has fields `money` of type Int, `payMe` of type MT (as we will #1 later this is a method), and in case it has a field `boss`, then the field is an object of type T. The type T:

$$\mu\ \alpha.\ \ll \texttt{money} : \mathsf{Int}, \texttt{payMe} : \mathrm{MT}, \texttt{boss} : \alpha? \gg \tag{2}$$

is an object type, and has a bound variable α that allows to refer to the whole type in the member types. In this case `boss` has type T. The type (2) has the same members of (1) with the same type.

Members in a type can be annotated with ? indicating a *potential* member, as in (1) which is the return type of function `Person`. When objects are created using function `Person` they are given type (1), which allows a member `boss` to be added later. This allows objects to evolve in a controlled manner. Note also, that the type of member `boss` is (2), that has the same members with the same type as (1). This captures the requirement that the boss of a person is also a person.

When a potential member is assigned to, it becomes *definite*, loosing its ?. To keep the type system manageable we only track assignments to variables (formal parameters and `this`) within the scope of a function. For the effect of assignments to variables to be visible outside a function they must be returned from the function with the appropriate type. For example, in function `employPerson`, the assignment `x.boss = y` makes `boss` definite in the type of `x`. The return type being $\ll \texttt{money} : \mathsf{Int}, \texttt{payMe} : \mathrm{MT}, \texttt{boss} : \mathrm{T} \gg$ in which `boss` has lost the annotation ?.

Functions are given types of the shape #1 where $\mathtt{t}_1$ is the type of the metavariable `this`, $\mathtt{t}_2$ the type of the parameter, and $\mathtt{t}_3$ the type of the return

value of the function. For instance, `moneyTrans` has type MT:

$$(\ll \texttt{money} : \mathsf{Int} \gg, \mathsf{Int}) \to \mathsf{Int}$$

saying that the metavariable `this` is $\ll$ `money` : Int $\gg$, the parameter is of type Int, and it returns an Int.

When a function is used as a method of an object upon calling we check that the receiver is a subtype of the declared type for `this` in the function. For example, with call `paul.payMe(10)` on line 26 we have that $\ll$ `money` : Int, `payMe` : MT, `boss` : T? $\gg$ is a subtype of $\ll$ `money` : Int $\gg$.

Subtyping, for object types, is based on the structures of the types concerned. For type t to be a subtype of type t′, t must declare at least the members defined in t′ with the same types. In case a member is definite in t′, then it must be definite also in t. We see this with the type, $\ll$ `money` : Int, `payMe` : MT, `boss` : T $\gg$, of variable `paul` being a subtype of $\ll$ `money` : Int $\gg$, the type of `this` declared in `moneyTrans`. This means that function `moneyTrans` can be used as a method of any object whose declared type contains member `money` of type Int.

Consider the definition of `moneyTrans1` that follows:

```
function moneyTrans1(x:Int):??
 {
  this:≪ money : Int ≫;
  this.money = this.money + x;
  moneyTrans1;
 }
```

`moneyTrans1` returns the function `moneyTrans1`. To give a type to this function we have to be able to refer in the return type of the function to the type of the whole function. To this extent we use, #1, the binder $\mu\,\alpha$, and give to `moneyTrans1` type:

$$\mu\ \alpha.(\mathsf{Int}, \mathsf{Int}) \to \alpha$$

The typed version of `moneyTrans1` can be written as follows:

```
function moneyTrans1(x:Int):μ α.(Int, Int) → α
 {
  this:≪ money : Int ≫;
  this.money = this.money + x;
  moneyTrans1;
 }
```

3 $\mathbf{JS}_0$

We have developed JS_0 a subset of JavaScript which includes the following features:

(i) functions used to create objects,

(ii) functions can be aliased and used as members of objects, and

```
1  function Person(x) {
2   this.money = x;
3   this.payMe = moneyTrans;
4   this
5  }
6
7  function moneyTrans(x) {
8   this.money = this.money + x;
9  }
10
11 function employPerson(x,y) {
12  x.boss = y; x
13 }
14
15 //Main
16 john = new Person(100);
17 paul = new Person(0);
18 paul = employPerson(paul,john); paul.payMe(10); paul.boss
```

Fig. 1. Untyped JS_0 Person Example

(iii) members can be added to objects dynamically.

We chose these features because, (i) represents the way objects are created in JavaScript, (ii) is a way by which objects acquire methods, and (iii) gives flexibility to programs.

JS_0 does not include the following JavaScript features: libraries of functions, native calls, global `this` (through a global object), dynamic variable creation, functions as objects, dynamic removal of members, delegation and prototyping. We omitted these features because the first three are not central to the paradigm, while the others are too difficult to support in a statically typed language. We can write the introductory examples from [10] in JS_0 assuming libraries of functions, and predefined types floats, strings, etc. The syntax of JS_0 is given in Figure 3. A program is a sequence of function declarations. In JS_0 functions may have only one formal parameter. The extension to functions with multiple parameters is trivial. For a program P, $\mathsf{P}(\mathsf{f})$ is defined as follows #1

3.1 Operational Semantics

We have a structural operational semantics for JS_0 that rewrites tuples of expressions, heaps and stacks into tuples of values, heaps and stacks in the context of a program. The signature of the rewriting relation $\twoheadrightarrow$ is:

$$\twoheadrightarrow : \mathit{Program} \rightarrow \mathit{Exp} \times \mathit{Heap} \times \mathit{Stack} \rightarrow (\mathit{Val} \cup \mathit{Dev}) \times \mathit{Heap} \times \mathit{Stack}$$

```
1  function Person(x:Int):≪ money : Int, payMe : MT, boss : T? ≫ {
2   this:≪ money : Int?, payMe : MT?, boss : T? ≫;
3   this.money = x;
4   this.payMe = moneyTrans;
5   this
6  }
7
8  function moneyTrans(x:Int):Int {
9   this:≪ money : Int ≫;
10  this.money = this.money + x
11 }
12
13 function employPerson(x:≪ money : Int, payMe : MT, boss : T? ≫,
14                        y:≪ money : Int, payMe : MT, boss : T? ≫):
15                           ≪ money : Int, payMe : MT, boss : T ≫
16 {
17  this:≪≫;
18  x.boss = y; x
19 }
20
21 //Main
22 john:≪ money : Int, payMe : MT, boss : T? ≫;
23 paul:≪ money : Int, payMe : MT, boss : T? ≫;
24 john = new Person(100);
25 paul = new Person(0);
26 paul=employPerson(paul,john);paul.payMe(10);paul.boss
```

Fig. 2. Typed JS_0 Person Example

where:

$$\begin{array}{lll} \mathsf{H} \in \mathit{Heap} & = & \mathit{Addr} \rightarrow_{\text{fin}} \mathit{Obj} \\ \mathsf{S} \in \mathit{Stack} & = & \#1 \\ \mathsf{v} \in \mathit{Val} & = & \{\texttt{null}\} \cup \mathit{FuncID} \cup \mathit{Addr} \cup \mathit{Int} \\ \mathsf{dv} \in \mathit{Dev} & = & \{\texttt{nullPntrExc}, \texttt{stuckErr}\} \\ \mathsf{o} \in \mathit{Obj} & = & \mathit{MemberID} \rightarrow_{\text{fin}} \mathit{Val} \end{array}$$

The heap maps addresses to objects, where addresses, *Addr*, are $\iota_0, ..\iota_n...$ We use $\rightarrow_{\text{fin}}$ to indicate a finite mapping. With $\mathsf{H}[\iota \mapsto \mathsf{v}]$ we denote the *updating of the value of the address* ι *in the heap* H *to* v. The stack maps `this` to an address and `x` to a value, where values, *Val*, are function identifiers (denoting functions), addresses (denoting objects), `null` , or integers. Finally objects are finite mappings between member identifiers and values. With $[\![\mathsf{m}_1 : \mathsf{v}_1 ... \mathsf{m}_n : \mathsf{v}_n]\!]$ we denote the object mapping m_i to v_i for $i \in 1 \cdots n$. For stacks and objects we use the updating notation previously defined for heaps.

To give a taste of the operational semantics, we show rule (mem-call). A

$\mathsf{P} \in \mathit{Program}$	::=	F^*	
$\mathsf{F} \in \mathit{FuncDecl}$	::=	`function` f (`x`) { e }	
$\mathsf{e} \in \mathit{Exp}$	::=	var	locals
		f	function identifier
		`new` f(e)	object creation
		e; e	sequence
		e.m(e)	member call
		e.m	member select
		f(e)	global call
		lhs = e	assignment
		e_1? e_2: e_3	conditional
		`null`	null
		`n`	integer
$\mathsf{var} \in \mathit{EnvVars}$	::=	`this` | `x`	
$\mathsf{lhs} \in \mathit{LeftSide}$	::=	`x` | e.m	

Identifiers

$\mathsf{f} \in \mathit{FuncID}$	::=	$\mathsf{f} \mid \mathsf{f}' \mid \ldots$	
$\mathsf{m} \in \mathit{MemberID}$	::=	$\mathsf{m} \mid \mathsf{m}' \mid \ldots$	

Fig. 3. Syntax of JS_0

full description of the rules is given in Appendix A.

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \iota, \mathsf{H}_1, \mathsf{S}_1 \\ \mathsf{e}_2, \mathsf{H}_1, \mathsf{S}_1 \twoheadrightarrow \mathsf{v}', \mathsf{H}_2, \mathsf{S}' \\ \mathsf{H}_2(\iota)(\mathsf{m}) = \mathsf{f} \\ \mathsf{P}(\mathsf{f}) = \texttt{function}\ \mathsf{f}(\texttt{x})\ \{\mathsf{e}'\} \\ \mathsf{e}', \mathsf{H}_2, \{\texttt{this} \mapsto \iota, \texttt{x} \mapsto \mathsf{v}'\} \twoheadrightarrow \mathsf{v}, \mathsf{H}', \mathsf{S}''\end{array}}{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2), \mathsf{H}, \mathsf{S} \twoheadrightarrow \mathsf{v}, \mathsf{H}', \mathsf{S}'}\ (\text{mem-call})$$

In rule (mem-call) we first evaluate the receiver and then the actual parameter of the method. We obtain the function definition (corresponding to the method) by looking up the value of member m in the receiver (obtained by evaluation of e) in P [5] . We execute the body with a stack in which `this` refers to the receiver of the call and `x` to the value of the actual parameter. Finally, we return the stack after the evaluation of the actual parameter and the heap resulting from the execution of the body of the method, so that `this` and `x` are bound to their value before the evaluation of the body, but the heap has the effects of the evaluation of the body.

Returning to the example in Figure 1, executing the body of function `Main` in the presence of an empty heap, H_0, with stack, S_0, mapping `john` and `paul` to `null` will produce heap H_1 and stack S_1 such that `john` and `paul` are mapped to ι_0 and ι_1 respectively, and

[5] For clearness of presentation we omit P from the reduction rules.

$\#1$
$\mathsf{H}_1(\iota_1) = [\![\texttt{money} : 10, \texttt{payMe} : \texttt{moneyTrans}, \texttt{boss} : \iota_0]\!]$

Note that the member `payMe` aliases function `moneyTrans` which was invoked when `paul.payMe(10)` was executed.

4 A Type System for $\mathbf{JS}_0$

In this section we introduce a fully-typed version of JS_0: $\mathrm{JS}_0^{\mathsf{T}}$.

4.1 Types

Figure 4 shows the parts of $\mathrm{JS}_0^{\mathsf{T}}$ that differ from JS_0 along with the definitions of types. We make the following observations:

- functions have a return type preceded by a colon,
- function formal parameters are given types, and
- function bodies start by declaring the type of the receiver, `this`.

Types t_1, ..., t_n, comprise object types, function types, or Int (the type of integers). Object types #1 list the methods and fields present in the object. We use the μ-binder to allow a type to refer to itself. So $\mu\ \alpha.\mathsf{M}$ where $\mathsf{M} =< < \mathsf{m}_1 : \mathsf{t}_1 \cdots \mathsf{m}_n : \mathsf{t}_n \gg$, is the type of an object with members m_1, ..., m_n of type t_1, ..., t_n, respectively. With $\mu\ \alpha.\mathsf{M}[\mathsf{m} \mapsto \mathsf{t}]$ we denote the *updating of the member* m *to type* t *in* M. In Figure 5 we define $\mathsf{M}(\mathsf{m})$, which selects the annotated type of the member m in M (if it is defined), and $\mathcal{T}(\mathsf{M}, \mathsf{m})$, which selects the type (without the annotation) of the member m in M. Function types #1. As for object types the μ-binder allows to refer to the whole type.

The definition of free variables of a type is the standard one:

$$\begin{array}{l}
\mathcal{FV}(\ll \mathsf{m}_1 : \mathsf{t}_1 ... \mathsf{m}_n : \mathsf{t}_n \gg) = \bigcup_{i \in 1...n} \mathcal{FV}(\mathsf{t}_i), \\
\mathcal{FV}((\mathsf{t}_1, \mathsf{t}_2) \to \mathsf{t}_3) = \bigcup_{i \in 1...3} \mathcal{FV}(\mathsf{t}_i), \\
\mathcal{FV}(\mu\ \alpha.\mathsf{t}) = \mathcal{FV}(\mathsf{t}) - \{\alpha\}, \\
\mathcal{FV}(\alpha) = \{\alpha\}, \quad \text{and} \quad \mathcal{FV}(\mathsf{Int}) = \emptyset.
\end{array}$$

We say that a type t is *well-formed* if it is closed ($\mathcal{FV}(\mathsf{t}) = \emptyset$), and, if the type is an object type, then it contains unique member definitions.

If the type of m is α, or M, or $\mu\ \alpha.\mathsf{M}$, or Int the member represents a field. In the case of α the type has the structure of the enclosing type ($\mu\ \alpha.\mathsf{M}$). If the type of m is $\mu\ \alpha.(\mathsf{t}_1, \mathsf{t}_2) \to \mathsf{t}_3$, or $(\mathsf{t}_1, \mathsf{t}_2) \to \mathsf{t}_3$, then m represents a method. The type of a function, f, in a program P is found using the *lookup function* $\mathcal{L}$ that follows:

$$\mathcal{L}(\mathsf{P}, \mathsf{f}) = \begin{cases} (\mathsf{t}, \mathsf{t}') \to \mathsf{t}'' & \text{if } \mathsf{P}(\mathsf{f}) = \texttt{function}\ \mathsf{f}(\mathsf{x} : \mathsf{t}') : \mathsf{t}''\{\ \texttt{this} : \mathsf{t};\ \mathsf{e}\} \\ \mathcal{U}df & \text{otherwise} \end{cases}$$

Syntax

$$
\begin{array}{lll}
\mathsf{P} \in \mathit{Program} & ::= & \mathsf{F}^* \\
\mathsf{F} \in \mathit{FuncDecl} & ::= & \texttt{function f (x:t'):t''} \ \{\ \texttt{this:t; e}\ \}
\end{array}
$$

Types

$$
\begin{array}{lll}
\mathsf{t} \in \mathit{Type} & ::= & \mu\ \alpha.\mathsf{M} \mid \mathsf{M} \mid \alpha \mid \mathsf{Int} \\
 & & \mu\ \alpha.(\mu\ \alpha.\mathsf{M}, \mathsf{t}) \to \mathsf{t} \mid (\mu\ \alpha.\mathsf{M}, \mathsf{t}) \to \mathsf{t} \\
\mathsf{M} \in \mathit{MemberTypes} & ::= & \ll (\mathsf{m} : \mathsf{t}[?])^* \gg \\
\alpha \in \mathit{ObjVar} & ::= & \alpha \mid \alpha' \mid \alpha'' \mid \ldots
\end{array}
$$

Fig. 4. Syntax of $\mathrm{JS}_0^{\mathsf{T}}$

$$
\mathsf{M}(\mathsf{m}) = \begin{cases} \mathsf{t} & \text{if } \mathsf{M} = \ll \cdots \mathsf{m} : \mathsf{t} \cdots \gg \\ \mathsf{t}? & \text{if } \mathsf{M} = \ll \cdots \mathsf{m} : \mathsf{t}? \cdots \gg \\ \mathcal{U}df & \text{otherwise} \end{cases}
$$

$$
\mathcal{T}(\mathsf{M}, \mathsf{m}) = \begin{cases} \mathsf{t} & \text{if } \mathsf{M}(\mathsf{m}) = \mathsf{t} \text{ or } \mathsf{M}(\mathsf{m}) = \mathsf{t}? \\ \mathcal{U}df & \text{if } \mathsf{M}(\mathsf{m}) = \mathcal{U}df \end{cases}
$$

Fig. 5. Selection of member's type

Some members of an object type are annotated with ?. This indicates a potential member, m:t ?, that can be assigned to later thus, allowing objects to evolve. In a well-typed program potential members may not be accessed until they have been assigned to.

4.1.1 Congruence and Subtyping

Congruence between types is defined in Fig. 6. With $\mathsf{t}_1[\alpha/\mathsf{t}_2]$, we denote the substitution of the free occurrences of α in t_1 with t_2. Object types are congruent up to α-conversion, permutation of their members, and unfolding of the bound variable, and function types are congruent up to α-conversion, and unfolding of the bound variable.

The *subtyping* judgement $\mathsf{t} \preccurlyeq \mathsf{t}'$, defined in Figure 7, means that an object or function of type t can be used whenever one of type t' is required.

For object types we have subtyping in width. Firstly, all definite members of M' must be present and congruent with those in M (first line of the rule for $\mu\ \alpha.\mathsf{M} \preccurlyeq \mu\ \alpha'.\mathsf{M}'$ in Figure 7). To ensure that the types are closed we substitute occurrences of bound variables by their enclosing type. The second condition

Reflexivity

$$\frac{}{\mathsf{t} \equiv \mathsf{t}}$$

Unfolding

$$\frac{}{\mu\ \alpha.\mathsf{t} \equiv \mathsf{t}[\alpha/\mu\ \alpha.\mathsf{t}]}$$

Alpha – conversion

$$\frac{\alpha' \notin\ \mathcal{FV}(\mathsf{t})}{\mu\ \alpha.\mathsf{t} \equiv \mu\ \alpha'.\mathsf{t}[\alpha/\alpha']}$$

Functions

$$\frac{\mathsf{t}_i \equiv \mathsf{t}'_i \quad i \in \{1,2,3\}}{\#1}$$

Reordering

$$\frac{\begin{array}{l}\forall\ \mathsf{m} \quad (\mathsf{M}(\mathsf{m}) = \mathsf{t} \iff \mathsf{M}'(\mathsf{m}) = \mathsf{t}') \quad \wedge \quad \mathsf{t}' \equiv \mathsf{t} \\ \forall\ \mathsf{m} \quad (\mathsf{M}(\mathsf{m}) = \mathsf{t}? \iff \mathsf{M}'(\mathsf{m}) = \mathsf{t}'?) \quad \wedge \quad \mathsf{t}' \equiv \mathsf{t}\end{array}}{\mu\ \alpha.\mathsf{M}\ \equiv\ \mu\ \alpha'.\mathsf{M}'}$$

Transitivity

$$\frac{\mathsf{t}_1 \equiv \mathsf{t}_2 \quad \mathsf{t}_2 \equiv \mathsf{t}_3}{\mathsf{t}_1 \equiv\ \mathsf{t}_3}$$

Fig. 6. Congruence for types

of the definition in Figure 7 (second line of the rule for $\mu\ \alpha.\mathsf{M} \preccurlyeq \mu\ \alpha'.\mathsf{M}'$) refers to the potential members in M'. In particular, it states that all potential members of M' must be present as potential or definite members of M with congruent types. This condition is needed to insure that the addition of a new member to an object does not break compatibility. #1

```
x:μ α. ≪ m1 : Int? ≫
y:μ α. ≪ m1 : α? ≫
z:μ α. ≪≫
z = x;
x.m1 = 5; .
y = z;
y.m1 = null
```

Any reasonable subtyping for object types is such that $\mu\ \alpha.\ \ll \mathsf{m1} : \mathsf{Int}? \gg$ and $\mu\ \alpha.\ \ll \mathsf{m1} : \alpha? \gg$ are subtypes of $\mu\ \alpha.\ \ll\gg$. So the assignment `z = x;` is correct. #1

For function types subtyping coincides with congruence. In future versions of this work we may relax this restriction and allow contravariance on the receiver and parameter type and covariance on the return type.

$$\frac{\mathsf{t} \equiv \mathsf{t}'}{\mathsf{t} \preccurlyeq \mathsf{t}'} \qquad \frac{\mathsf{t}_1 \preccurlyeq \mathsf{t}_2 \quad \mathsf{t}_2 \preccurlyeq \mathsf{t}_3}{\mathsf{t}_1 \preccurlyeq \mathsf{t}_3}$$

$$\frac{\begin{array}{l}\forall\ \mathsf{m} \quad (\mathsf{M}'(\mathsf{m}) = \mathsf{t}' \implies \mathsf{M}(\mathsf{m}) = \mathsf{t}) \quad \wedge \quad \mathsf{t}'[\alpha'/\mu\ \alpha'.\mathsf{M}'] \equiv \mathsf{t}[\alpha/\mu\ \alpha.\mathsf{M}]) \\ \forall\ \mathsf{m} \quad (\mathsf{M}'(\mathsf{m}) = \mathsf{t}'? \implies \mathcal{T}(\mathsf{M}, \mathsf{m}) = \mathsf{t}) \quad \wedge \quad \mathsf{t}'[\alpha'/\mu\ \alpha'.\mathsf{M}'] \equiv \mathsf{t}[\alpha/\mu\ \alpha.\mathsf{M}])\end{array}}{\mu\ \alpha.\mathsf{M} \preccurlyeq \mu\ \alpha'.\mathsf{M}'}$$

Fig. 7. Subtyping

Given types t and t' it is decidable whether $\mathsf{t} \preccurlyeq \mathsf{t}'$ or not.

4.2 Typing of Expressions

Typing an expression e in the context of a program P, and environment Γ has the form:

$$\mathsf{P}, \Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma'$$

The environment, $\Gamma = \{\texttt{this} : \mu\ \alpha.\mathsf{M}, \texttt{x} : \mathsf{t}\}$, maps the receiver, `this`, to a well-formed object type, and the formal parameter, `x`, to a well-formed type. With $\Gamma[\mathsf{var} \mapsto \mathsf{t}]$ we denote the *updating of* var *to type* t *in* Γ. The environment on the right hand side of the typing judgement reflects the changes to the type of the receiver or parameter while typing the expression. The only change possible is the removal of ? from the type of a member of `this` or `x`.

Consider the typing rules of Figure 8. Rules (*var*), (*func*), and (*const*) are straightforward. #1

In rule $(mem - acc)$ the expression e must be of an object type in which the member m is definite, i.e. defined and without the annotation ?. In the type of the member, all occurrences of α are substituted with the enclosing type to return a closed type.

In rule $(meth - call)$ we check that the type of the receiver is an object type in which the member m has a function type, and it is definite. Moreover, the type of the receiver and actual parameter must be subtypes of the declared type of the receiver and formal parameter for the function f. #1

In (*call*) we consider global calls and constructors, and require that the type of the receiver defined in the function has no definite members. This is consistent with the operational semantics, as in the case of global call and object creation we start with an empty receiver object.

In rule $(assign - add)$ we modify the type environment, by removing from the type of member m (of `this` or of the formal parameter) the annotation ? (if the member has a ? annotation). From this point the member m may be accessed. The type of the expression assigned must be a subtype of the type

of the member after being closed. #1

$$\mathsf{P}, \Gamma \vdash \mathtt{x}.\mathsf{m}_2 = \mathtt{x} : \mathsf{t} \parallel \Gamma'$$

where Γ' maps `this` to $\Gamma(\mathtt{this})$ and `x` to

$$\ll \mathsf{m}_1 : \mathsf{Int}, \mathsf{m}_2 : \mu\ \alpha.\ \ll \mathsf{m}_1 : \mathsf{Int}, \mathsf{m}_2 : \alpha? \gg\gg$$

This reflects the updating of member m_2. Note that the type of the member m_2 of the type of m_2 has still the annotation ?.

The rule $(assign - upd)$, which assumes that the member m be defined is similar.

In rule (*cond*) the operation $\mathsf{t} \sqcup \mathsf{t}'$ is applicable (and so also the rule) only when *the types* t *and* t' *are compatible*, that is:

- either $\mathsf{t} \equiv \mathsf{t}'$,
- or $\mathsf{t} \equiv \mu\ \alpha.\mathsf{M}$, $\mathsf{t}' \equiv \mu\ \alpha'.\mathsf{M}'$ and
  - $\mathsf{M}(\mathsf{m}) \neq \mathcal{U}df$ if and only if $\mathsf{M}'(\mathsf{m}) \neq \mathcal{U}df$ and
  - $\mathcal{T}(\mathsf{M}, \mathsf{m}) = \mathsf{t}''$ implies $\mathcal{T}(\mathsf{M}', \mathsf{m}) = \mathsf{t}'''$ and $\mathsf{t}''[\alpha/\mathsf{t}] \equiv \mathsf{t}'''[\alpha'/\mathsf{t}']$.

Therefore, for object types the types must have the same members (that could be definite in one type and potential in the other) with congruent types. If t and t' are compatible define *the upper bound of* t *and* t', $\mathsf{t} \sqcup \mathsf{t}'$ by:

$$\mathsf{t} \sqcup \mathsf{t}' = \begin{cases} \mathsf{t} & \text{if } \mathsf{t}' \equiv \mathsf{t} \\ \mu\ \alpha.\mathsf{M}'' & \text{if } \mathsf{t} \equiv \ \mu\ \alpha.\mathsf{M},\ \mathsf{t}' \equiv \ \mu\ \alpha'.\mathsf{M}' \text{ where} \\ & \quad \mathsf{M}''(\mathsf{m}) = \mathsf{t}'' \ \#1\ (\mathsf{M}(\mathsf{m}) = \mathsf{t}'' \text{ and } \mathsf{M}'(\mathsf{m}) = \mathsf{t}''') \\ & \quad \mathsf{M}''(\mathsf{m}) = \mathsf{t}''? \ \#1\ (\mathsf{M}(\mathsf{m}) = \mathsf{t}''? \text{ or } \mathsf{M}'(\mathsf{m}) = \mathsf{t}'''?) \end{cases}$$

For object types a member of $\mathsf{t} \sqcup \mathsf{t}'$ is definite if it is a definite member of both t and t', otherwise it is a potential member.

Compatibility for environments is defined as follows: Γ and Γ' are compatible if and only if for all var such that $\mathsf{var} \in \Gamma$, and $\mathsf{var} \in \Gamma'$, $\Gamma(\mathsf{var})$ is compatible with $\Gamma'(\mathsf{var})$. If Γ and Γ' are compatible

$$\Gamma \sqcup \Gamma' = \{\mathtt{this} : \Gamma(\mathtt{this}) \sqcup \Gamma'(\mathtt{this}), \mathtt{x} : \Gamma(\mathtt{x}) \sqcup \Gamma'(\mathtt{x})\}$$

We can prove that if $\mathsf{P}, \Gamma \vdash \mathsf{e}_1 : \mathsf{t} \parallel \Gamma'$, then Γ and Γ' are compatible, since Γ' may differ from Γ only because some potential members of a var have become definite. Therefore, if $\mathsf{P}, \Gamma \vdash \mathsf{e}_1 : \mathsf{t} \parallel \Gamma'$ and $\mathsf{P}, \Gamma \vdash \mathsf{e}_2 : \mathsf{t}' \parallel \Gamma''$ then Γ'' and Γ' are compatible. Clearly, if there is no relation between e_1 and e_2, there may be no relation between t and t' in which case rule (*cond*) is not applicable.

In rule $(var - ass)$ the type of e has to be a subtype of the type of $\mathtt{x}$ in Γ'.

The operation on types, $\mathsf{t}' \star \mathsf{t}$, which is defined if $\mathsf{t} \preccurlyeq \mathsf{t}'$, is as follows:

$$\mathsf{t}' \star \mathsf{t} = \begin{cases} \mathsf{t}' & \text{if } \mathsf{t}' \equiv \mathsf{t} \\ \mu\ \alpha'.\mathsf{M}'' & \text{if } \mathsf{t} \equiv\ \mu\ \alpha.\mathsf{M},\ \mathsf{t}' \equiv\ \mu\ \alpha'.\mathsf{M}' \text{ where} \\ & \mathsf{M}''(\mathsf{m}) = \mathsf{t}''\ \#1 \\ & \mathsf{M}''(\mathsf{m}) = \mathsf{t}''\ ?\#1 \end{cases}$$

When applied to object types, $\mathsf{t}' \star \mathsf{t}$ returns a type with all the members of t', with those that are definite in t being made definite in $\mathsf{t}' \star \mathsf{t}$ (Note that since $\mathsf{t} \preccurlyeq \mathsf{t}'$ any definite member of t' must be also a definite member of t). In the rule $(var - ass)$ this operation is used to remove the annotation ? from the members of the type of $\mathtt{x}$ in Γ' (if the type is an object type) which are definite members of the type of e. #1

#1

A program P is *well-formed* if all the function declarations in P are well-typed, that is: if f is such that $\mathsf{P}(\mathsf{f}) = \mathtt{function}\ \mathsf{f}(\mathtt{x} : \mathsf{t}') : \mathsf{t}''\{\ \mathtt{this} : \mathsf{t};\ \mathsf{e}\}$

- the types t, t' and t'' are well-formed, $\mathsf{t} \equiv \mu\ \alpha.\mathsf{M}$,
- $\mathsf{P}, \{\mathtt{this} : \mu\ \alpha.\mathsf{M}, \mathtt{x} : \mathsf{t}'\} \vdash \mathsf{e} : \mathsf{t}''' \parallel \Gamma'$ and $\mathsf{t}''' \preccurlyeq \mathsf{t}''$.

5 Formal Properties of the Type System

In this section we outline the proof that our type system is sound w.r.t. to the operational semantics given in Section 3.1. We assume that types are well-formed. We first define the notion of a value being compatible with a given type. The definition is given co-inductively by first defining the properties that any agreement relation #1 should have.

Definition 5.1 Given a heap, H, and a #1 program, P, we say that $\mathsf{A} \subseteq (\mathit{Val} \times \mathit{Type})$ is an *agreement relation* if:

- $(\mathtt{null}, \mathsf{t}) \in \mathsf{A}$ if and only if #1
- $(\mathtt{n}, \mathsf{t}) \in \mathsf{A}$ if and only if $\mathsf{t} = \mathsf{Int}$,
- if $(\mathsf{f}, \mathsf{t}) \in \mathsf{A}$, then #1,
- if $(\iota, \mathsf{t}) \in \mathsf{A}$, then $\mathsf{t}\#1\mu\ \alpha.\mathsf{M}$, and
  - $\mathsf{H}(\iota) = [\![\mathsf{m}_1 : \mathsf{v}_1 \ldots \mathsf{m}_p : \mathsf{v}_p]\!]$
  - for all m and t' such that $\mathsf{M}(\mathsf{m}) = \mathsf{t}'$ we have that
    $\mathsf{m} = \mathsf{m}_i$ for some $i \in 1...p$ and $(\mathsf{v}_i, \mathsf{t}'[\alpha/\mathsf{t}]) \in \mathsf{A}$
  - for all m and t' such that $\mathsf{M}(\mathsf{m}) = \mathsf{t}'?$
    if $\mathsf{m} = \mathsf{m}_i$ for some $i \in 1...p$ then $(\mathsf{v}_i, \mathsf{t}'[\alpha/\mathsf{t}]) \in \mathsf{A}$

If A and A' are agreement relations also $\mathsf{A} \cup \mathsf{A}'$ is an agreement relation. Therefore given a heap, H, and a program, P, the union of all agreement relations defines the relation between values and types, that says when a value has a given type.

$$\frac{}{\mathsf{P},\Gamma \vdash \mathtt{this} : \Gamma(\mathtt{this}) \parallel \Gamma}\ (var) \qquad \frac{\#1}{\mathsf{P},\Gamma \vdash \mathsf{f} : \mathsf{t} \parallel \Gamma}\ (func)$$
$$\mathsf{P},\Gamma \vdash \mathtt{x} : \Gamma(\mathtt{x}) \parallel \Gamma$$

$$\frac{}{\mathsf{P},\Gamma \vdash \mathtt{null} : \mu\ \alpha.\mathsf{M} \parallel \Gamma}\ (const)$$
$$\mathsf{P},\Gamma \vdash \mathtt{n} : \mathsf{Int} \parallel \Gamma$$

$$\frac{\mathsf{P},\Gamma \vdash \mathsf{e} : \mu\ \alpha.\mathsf{M} \parallel \Gamma' \quad \mathsf{M}(\mathsf{m}) = \mathsf{t}}{\mathsf{P},\Gamma \vdash \mathsf{e.m} : \mathsf{t}[\alpha/\mu\ \alpha.\mathsf{M}] \parallel \Gamma'}\ (mem-acc) \qquad \frac{\mathsf{P},\Gamma \vdash \mathsf{e}_1 : \mathsf{t} \parallel \Gamma' \quad \mathsf{P},\Gamma' \vdash \mathsf{e}_2 : \mathsf{t}' \parallel \Gamma''}{\mathsf{P},\Gamma \vdash \mathsf{e}_1; \mathsf{e}_2 : \mathsf{t}' \parallel \Gamma''}\ (seq)$$

$$\frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e}_1 : \mu\ \alpha.\mathsf{M} \parallel \Gamma' \\ \mathsf{M}(\mathsf{m}) = \mu\ \alpha'.(\mathsf{t}_1,\mathsf{t}_1') \to \mathsf{t}_1'' \\ \mathsf{P},\Gamma' \vdash \mathsf{e}_2 : \mathsf{t} \parallel \Gamma'' \\ \mathsf{t} \preccurlyeq \mathsf{t}_1'[\alpha'/\mathsf{M}(\mathsf{m})] \\ \mu\ \alpha.\mathsf{M} \preccurlyeq \mathsf{t}_1[\alpha'/\mathsf{M}(\mathsf{m})]\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{e}_1.\mathsf{m}(\mathsf{e}_2) : \mathsf{t}_1''[\alpha'/\mathsf{M}(\mathsf{m})] \parallel \Gamma''}\ (meth-call) \qquad \frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma' \\ \mathcal{L}(\mathsf{P},\mathsf{f}) = \ \mu\ \alpha'.(\mu\ \alpha.\mathsf{M},\mathsf{t}_1') \to \mathsf{t}_1'' \\ \mathsf{t} \preccurlyeq \mathsf{t}_1'[\alpha'/\mathcal{L}(\mathsf{P},\mathsf{f})] \\ \{\mathsf{t}' \mid \mathsf{M}(\mathsf{m}) = \mathsf{t}'\} = \emptyset\end{array}}{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathtt{new}\ \mathsf{f}(\mathsf{e}) : \mathsf{t}_1''[\alpha'/\mathcal{L}(\mathsf{P},\mathsf{f})] \parallel \Gamma' \\ \mathsf{P},\Gamma \vdash \mathsf{f}(\mathsf{e}) : \mathsf{t}_1''\#1 \parallel \Gamma'\end{array}}\ (call)$$

$$\frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e}_2 : \mathsf{t} \parallel \Gamma' \\ \Gamma'(\mathsf{var}) = \mu\ \alpha.\mathsf{M} \\ \mathsf{t}' = \mathsf{M}[\alpha/\mu\ \alpha.\mathsf{M}] \\ \mathcal{T}(\mathsf{t}',\mathsf{m}) = \mathsf{t}'' \\ \mathsf{t} \preccurlyeq \mathsf{t}'' \\ \Gamma'' = \Gamma'[\mathsf{var} \mapsto \mathsf{t}'[\mathsf{m} \mapsto \mathsf{t}'']]\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{var.m} = \mathsf{e}_2 : \mathsf{t} \parallel \Gamma''}\ (assign-add) \qquad \frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e}_1 : \mu\ \alpha.\mathsf{M} \parallel \Gamma' \\ \mathsf{M}(\mathsf{m}) = \mathsf{t}' \\ \mathsf{P},\Gamma' \vdash \mathsf{e}_2 : \mathsf{t} \parallel \Gamma'' \\ \mathsf{t} \preccurlyeq \mathsf{t}'[\alpha/\mu\ \alpha.\mathsf{M}]\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{e}_1.\mathsf{m} = \mathsf{e}_2 : \mathsf{t} \parallel \Gamma''}\ (assign-upd)$$

$$\frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e}_1 : \mathsf{Int} \parallel \Gamma' \\ \mathsf{P},\Gamma' \vdash \mathsf{e}_2 : \mathsf{t} \parallel \Gamma'' \\ \mathsf{P},\Gamma' \vdash \mathsf{e}_3 : \mathsf{t}' \parallel \Gamma'''\end{array}}{\mathsf{P},\Gamma \vdash \mathsf{e}_1?\ \mathsf{e}_2 :\ \mathsf{e}_3 : \mathsf{t} \sqcup \mathsf{t}' \parallel \Gamma'' \sqcup \Gamma'''}\ (cond) \qquad \frac{\begin{array}{l}\mathsf{P},\Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma' \\ \mathsf{t} \preccurlyeq \Gamma'(\mathtt{x}) \\ \Gamma'' = \Gamma'[\mathtt{x} \mapsto \Gamma'(\mathtt{x}) \star \mathsf{t}]\end{array}}{\mathsf{P},\Gamma \vdash \mathtt{x} = \mathsf{e} : \mathsf{t} \parallel \Gamma''}\ (var-ass)$$

Fig. 8. Typing rules for expressions in $\mathrm{JS}_0^{\mathsf{T}}$

Definition 5.2 *Value* v *is compatible with type* t *in* H, $\mathsf{P},\mathsf{H} \vdash \mathsf{v} \blacktriangleleft \mathsf{t}$, if for some agreement relation A on H and P we have that $(\mathsf{v},\mathsf{t}) \in \mathsf{A}$

Note that an address may be compatible with more than one type. In particular, if a value is compatible with a type, then it is compatible with all its supertypes.

Lemma 5.3 *If* $\mathsf{t} \preccurlyeq \mathsf{t}'$ *and* $\mathsf{P},\mathsf{H} \vdash \mathsf{v} \blacktriangleleft \mathsf{t}$ *then* $\mathsf{P},\mathsf{H} \vdash \mathsf{v} \blacktriangleleft \mathsf{t}'$

In the following we define when *a stack* S *and a heap* H *are compatible with an environment* Γ.

Definition 5.4 $\mathsf{P},\Gamma \vdash \mathsf{H},\mathsf{S} \diamond$ holds if $\mathsf{P},\mathsf{H} \vdash \mathsf{S}(\texttt{this}) \blacktriangleleft \Gamma(\texttt{this})$ and $\mathsf{P},\mathsf{H} \vdash \mathsf{S}(\mathtt{x}) \blacktriangleleft \Gamma(\mathtt{x})$

We introduce a relation between pairs of heaps, stacks saying that a pair heap, stack can be obtained from the other during the evaluation of an expression.

Definition 5.5 Given heaps H and H', and stacks S and S', $\mathsf{P} \vdash \mathsf{H},\mathsf{S} \rhd \mathsf{H}',\mathsf{S}'$ holds if:

- $\mathsf{S}(\texttt{this}) = \mathsf{S}'(\texttt{this})$.
- for all types t if $\mathsf{P},\mathsf{H} \vdash \mathsf{S}(\mathtt{x}) \blacktriangleleft \mathsf{t}$ holds then also $\mathsf{P},\mathsf{H} \vdash \mathsf{S}'(\mathtt{x}) \blacktriangleleft \mathsf{t}$ holds.
- for all addresses ι and types t if $\mathsf{P},\mathsf{H} \vdash \iota \blacktriangleleft \mathsf{t}$ holds then also $\mathsf{P},\mathsf{H}' \vdash \iota \blacktriangleleft \mathsf{t}$ holds.

We can now state the main lemma.

Lemma 5.6 *For a well-formed program* P*, environment* Γ*, and expression* e*, such that:*

$$\mathsf{P},\Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma'$$

If $\mathsf{e},\mathsf{H},\mathsf{S} \twoheadrightarrow \mathsf{v},\mathsf{H}',\mathsf{S}'$*, and* $\mathsf{P},\Gamma \vdash \mathsf{H},\mathsf{S} \diamond$ *then*

(i) $\mathsf{P},\mathsf{H}' \vdash \mathsf{v} \blacktriangleleft \mathsf{t}$,

(ii) $\mathsf{P},\Gamma' \vdash \mathsf{H}',\mathsf{S}' \diamond$, *and*

(iii) $\mathsf{P} \vdash \mathsf{H},\mathsf{S} \rhd \mathsf{H}',\mathsf{S}'$

The soundness theorem asserts that if an expression is well-typed in a type environment Γ, then the evaluation of the expression starting in a heap and stack that agree with Γ cannot produce a run-time error. That is, the result of the evaluation is either a value of the right type, or it is a `nullPntrExc` exception. In particular, it is not a `stuckErr` error.

Theorem 5.7 *[Type Soundness] For a well-formed program* P*, environment* Γ*, and expression* e*, such that:*

$$\mathsf{P},\Gamma \vdash \mathsf{e} : \mathsf{t} \parallel \Gamma'$$

If $\mathsf{P},\Gamma \vdash \mathsf{H},\mathsf{S} \diamond$ *and* $\mathsf{e},\mathsf{H},\mathsf{S} \twoheadrightarrow \mathsf{w},\mathsf{H}',\mathsf{S}'$*, then*

- *either* $\mathsf{w} = \mathsf{v}$, *and* $\mathsf{P}, \mathsf{H}' \vdash \mathsf{v} \blacktriangleleft \mathsf{t}$,
- *or* $\mathsf{w} = \texttt{nullPntrExc}$.

6 Comparisons and Future Work

In this paper a flexible type system for an idealized version of JavaScript is defined, and its soundness outlined. JavaScript is an object based language allowing extensible objects, and sharing of method bodies.

Type systems for object based languages have been developed mainly in a functional setting, see [1] and [9]. An imperative type safe object oriented language, TOIL, was introduced in [6]. Even though the language is class based, its type system does not identify types with classes. This makes the definition of types similar to ours. TOIL, however, does not have extensible objects, so there is no need for identifying potential members.

Extensible objects have being considered in a functional setting in [8]. An imperative calculus for extensible objects was proposed by Bono and Fisher, in [5]. In Bono and Fisher type system there are two types for objects: the *proto*-types that can be extended and the *object*-types that cannot. The type system tracks potential members. The main difference between our system and Bono Fisher type system is the fact that we use recursive types (instead of row types plus universal and existential quantification). This makes possible, for us, to have a decidable type inference algorithm, see the final paragraph of this section. Note that, Bono and Fisher's aim was to encode classes in their object calculus, not to obtain a type inference algorithm. Recursive types with subtyping have been studied in conjunction with functional programming languages by various researchers, see for instance [3].

Alias types are used in [4] and [7] to track the evolution of objects. In particular, in [7] potential members are used for the same purpose as the current paper. Alias types are, however, very different from the types used in this paper. They are singleton types identified with the address of objects.

The need for ensuring type safety in dynamically typed languages has been widely recognized. See for instance [11], [2], and [13]. In these papers, constraints are defined that insure that terms for which the inferred constraints are solvable do not cause message not understood errors. We approach the same problem differently, we first define a type system, that has good properties, such as soundness (well-typed expressions do not cause message not understood errors), and expressiveness (all the significant examples we have can be typed). Our next step will be defining a type inference algorithm such that the type inferred for a term is a type derivable for the term. In particular, we would like to achieve a principal typing, that is a typing from which all the typing of a term be derivable. Principality insures that we can do the type analysis in a modular way, that is we can type check two expressions separately and then type check their composition just based on their type information. This is not possible for the systems of [11], [2], and [13].

Acknowledgements

We would like to thank Sophia Drossopoulou for very detailed comments and useful suggestions, and Mario Coppo for his help and insight. We would also like to thank our colleagues at Imperial College Department of Computing and Dipartimento di Informatica of Torino University.

References

[1] Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, NY, 1996.

[2] Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. *Lecture Notes in Computer Science*, 707:247–262, 1993.

[3] Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.

[4] C. Anderson, F. Barbanera, M. Dezani-Ciancaglini, and S. Drossopoulou. Can addresses be types? (a case study: objects with delegation). In *WOOD '03*, volume 82 of *ENTCS*. Elsevier, 2003.

[5] V. Bono and K. Fisher. An Imperative, First-Order Calculus with Object Extension. In *Proc. of ECOOP'98*, volume 1445 of *LNCS*, pages 462–497, 1998. A preliminary version already appeared in Proc. of 5th Annual FOOL Workshop.

[6] Kim Bruce, A. Schuett, and R. van Gent. Polytoil: A type safe polymorphic object-oriented language. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP)*, 1995.

[7] F. Damiani and P. Giannini. Alias types for environment aware computations. In *WOOD '03*, volume 82 of *ENTCS*. Elsevier, 2003.

[8] K. Fisher. *Type Systems for Object-Oriented Programming Languages*. PhD thesis, Stanford University, 1996. Available as Stanford Computer Science Technical Report number STAN-CS-TR-98-1602.

[9] K. Fisher, F. Honsell, and J. C. Mitchell. A Lambda Calculus of Objects and Method Specialization. *Nordic Journal of Computing*, 1(1):3–37, 1994. A preliminary version appeared in *Proc. of IEEE Symp. LICS'93*.

[10] David Flanagan. *JavaScript - The Definitive Guide*. O'Reilly, 1998.

[11] Justin O. Graver and Ralph E. Johnson. A type system for smalltalk. In *Proceedings of the 17th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 136–150. ACM Press, 1990.

[12] Netscape. *JavaScript Language Reference*. http://developer.netscape.com/docs/manuals/js/.

[13] Mike Salib. Static type inference (for python) with starkiller. http://www.python.org/pycon/dc2004/papers/1/paper.pdf, 2004.

A Operation Semantics of JS_0

In Figures A.1, A.2 and A.3 we list the rules of the operational semantics for JS_0. The operational semantics rewrites tuples of expressions, heaps and stacks into tuples of values, heaps and stacks in the context of a program, P. The signature of the rewriting relation and the definitions of the components are given in Section 3.1.

In Figure A.1 we give the rules that describe execution when there are no errors. We discuss the most interesting rules, namely: (var), (mem-sel), (param-ass), (new), (cond-true), (cond-false). (Rule (mem-call) was discussed in the Section 3.1).

In (var) the receiver, `this`, or parameter, `x`, are looked up in the stack, and heap and stack are unmodified.

In (mem-sel) member m is looked up in the receiver ι (obtained by evaluation of e) in the heap. If m is not found in ι, then this rule is not applicable (instead one of the rules generating errors can be applied).

In (param-ass) we replace the value of `x` in the stack with the value obtained by execution of e.

In (new) we evaluate the body of function f (looked up in P) with a stack mapping `this` to a fresh address that points to an empty object, and the formal parameter, `x`, to the value obtained by the evaluation of the actual parameter.

In (cond-true) and (cond-false) we evaluate the conditional test, e_1. If the value is 0 then we return the result of the evaluation of e_2, and if the value is an integer greater than 0 we return the result of the evaluation of e_3.

A.0.1 Runtime Errors

Figure A.2 gives rules for the cases where something has gone wrong. The possible errors are: access to members of null objects, access to non existing members of objects or non existing objects, undefined functions, the test of a conditional expression is not evaluated to an integer. The first error raises `nullPntrExc`, whereas the others result in `stuckErr`. Our type system ensure that a well-typed expression cannot evaluate to `stuckErr`.

Figure A.3 gives the rules for propagation of exceptions and errors once they have been generated. An exception or error is propagated upwards until it reaches the top, as with Java exceptions.

#1 #1

$$\frac{\mathsf{e}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \iota, \mathsf{H}', \mathsf{S}'}{\mathsf{e.m}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{H}'(\iota)(\mathsf{m}), \mathsf{H}', \mathsf{S}'}\ (\text{mem-sel}) \qquad \frac{\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}', \mathsf{H}_1, \mathsf{S}_1 \quad \mathsf{e}_2, \mathsf{H}_1, \mathsf{S}_1 \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}'}{\mathsf{e}_1; \mathsf{e}_2, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}'}\ (\text{seq})$$

$$\frac{\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \iota, \mathsf{H}_1, \mathsf{S}_1 \quad \mathsf{e}_2, \mathsf{H}_1, \mathsf{S}_1 \twoheadrightarrow \ \mathsf{v}, \mathsf{H}_2, \mathsf{S}' \quad \mathsf{H}' = \mathsf{H}_2[\iota \mapsto \mathsf{H}_2(\iota)[\mathsf{m} \mapsto \mathsf{v}]]}{\mathsf{e}_1.\mathsf{m} = \mathsf{e}_2, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}'}\ (\text{mem-ass}) \qquad \frac{\mathsf{e}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}'' \quad \mathsf{S}' = \mathsf{S}[\mathtt{x} \mapsto \mathsf{v}]}{\mathtt{x} = \mathsf{e}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}'}\ (\text{param-ass})$$

$$\frac{\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}', \mathsf{H}'', \mathsf{S}'' \quad \mathsf{v}' > 0 \quad \mathsf{e}_2, \mathsf{H}'', \mathsf{S}'' \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}'}{\mathsf{e}_1 ?\ \mathsf{e}_2 :\ \mathsf{e}_3, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}'}\ (\text{cond-true}) \qquad \frac{\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ 0, \mathsf{H}'', \mathsf{S}'' \quad \mathsf{e}_3, \mathsf{H}'', \mathsf{S}'' \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}'}{\mathsf{e}_1 ?\ \mathsf{e}_2 :\ \mathsf{e}_3, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}'}\ (\text{cond-false})$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}', \mathsf{H}_1, \mathsf{S}' \\ \mathsf{P}(\mathsf{f}) = \mathtt{function}\ \mathsf{f}(\mathtt{x} : \mathsf{t}') : \mathsf{t}''\{\ \mathtt{this} : \mathsf{t};\ \mathsf{e}\} \\ \iota \text{ is new in } \mathsf{H}_1 \text{ and } \mathsf{H}_2 = \mathsf{H}_1[\iota \mapsto [\![]\!]] \\ \mathsf{e}', \mathsf{H}_2, \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}'\} \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}''\end{array}}{\mathtt{new}\ \mathsf{f}(\mathsf{e}), \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \iota, \mathsf{H}', \mathsf{S}'}\ (\text{new})$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \iota, \mathsf{H}_1, \mathsf{S}_1 \\ \mathsf{e}_2, \mathsf{H}_1, \mathsf{S}_1 \twoheadrightarrow \ \mathsf{v}', \mathsf{H}_2, \mathsf{S}' \\ \mathsf{H}_2(\iota)(\mathsf{m}) = \mathsf{f} \\ \mathsf{P}(\mathsf{f}) = \mathtt{function}\ \mathsf{f}(\mathtt{x} : \mathsf{t}') : \mathsf{t}''\{\ \mathtt{this} : \mathsf{t};\ \mathsf{e}\} \\ \mathsf{e}', \mathsf{H}_2, \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}'\} \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}''\end{array}}{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2), \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}'}\ (\text{mem-call})$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}', \mathsf{H}_1, \mathsf{S}' \\ \mathsf{P}(\mathsf{f}) = \mathtt{function}\ \mathsf{f}(\mathtt{x} : \mathsf{t}') : \mathsf{t}''\{\ \mathtt{this} : \mathsf{t};\ \mathsf{e}\} \\ \iota \text{ is new in } \mathsf{H}_1 \text{ and } \mathsf{H}_2 = \mathsf{H}_1[\iota \mapsto [\![]\!]] \\ \mathsf{e}', \mathsf{H}_2, \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}\} \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}''\end{array}}{\mathsf{f}(\mathsf{e}), \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}, \mathsf{H}', \mathsf{S}'}\ (\text{func-call})$$

Fig. A.1. Operational Semantics of JS_0

$$\frac{\mathsf{e}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \texttt{null}, \mathsf{H}', \mathsf{S}'}{\begin{array}{l}\mathsf{e.m}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \texttt{nullPntrExc}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e.m} = \mathsf{e}', \mathsf{H}, \mathsf{S} \twoheadrightarrow \texttt{nullPntrExc}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e.m}(\mathsf{e}'), \mathsf{H}, \mathsf{S} \twoheadrightarrow \texttt{nullPntrExc}, \mathsf{H}', \mathsf{S}'\end{array}}$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \mathsf{v}, \mathsf{H}', \mathsf{S}' \\ \mathsf{v} \neq \texttt{null} \\ \mathsf{v} \notin \mathit{Addr} \text{ or } (\mathsf{v} = \iota \text{ and } \mathsf{H}(\iota) = \mathcal{U}\mathit{df})\end{array}}{\begin{array}{l}\mathsf{e.m}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e.m} = \mathsf{e}', \mathsf{H}, \mathsf{S} \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e.m}(\mathsf{e}'), \mathsf{H}, \mathsf{S} \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \mathsf{S}'\end{array}}$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \iota, \mathsf{H}', \mathsf{S}' \\ \mathsf{H}'(\iota)(\mathsf{m}) = \mathcal{U}\mathit{df}\end{array}}{\begin{array}{l}\mathsf{e.m}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e.m} = \mathsf{e}', \mathsf{H}, \mathsf{S} \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e.m}(\mathsf{e}'), \mathsf{H}, \mathsf{S} \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \mathsf{S}'\end{array}}$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \iota, \mathsf{H}_1, \mathsf{S}_1 \\ \mathsf{e}_2, \mathsf{H}_1, \mathsf{S}_1 \twoheadrightarrow \mathsf{v}', \mathsf{H}', \mathsf{S}' \\ \mathsf{H}'(\iota)(\mathsf{m}) = \mathsf{f} \\ \mathsf{P}(\mathsf{f}) = \mathcal{U}\mathit{df}\end{array}}{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2), \mathsf{H}, \mathsf{S} \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \mathsf{S}'}$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \mathsf{v}', \mathsf{H}', \mathsf{S}' \\ \mathsf{P}(\mathsf{f}) = \mathcal{U}\mathit{df}\end{array}}{\begin{array}{l}\mathsf{f}(\mathsf{e}), \mathsf{H}, \mathsf{S} \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \mathsf{S}' \\ \#1\end{array}}$$

$$\frac{\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \iota, \mathsf{H}', \mathsf{S}'}{\mathsf{e}_1 ?\ \mathsf{e}_2 :\ \mathsf{e}_3, \mathsf{H}, \mathsf{S} \twoheadrightarrow \texttt{stuckErr}, \mathsf{H}', \mathsf{S}'}$$

Fig. A.2. Operational Semantics - generation of exceptions

$$\frac{\mathsf{e}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}'}{\begin{array}{l}\mathtt{x} = \mathsf{e}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}' \\ \mathsf{f}(\mathsf{e}), \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}' \\ \ \mathbf{new}\ \ \mathsf{f}(\mathsf{e}), \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e}.\mathsf{m}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e}.\mathsf{m} = \mathsf{e}', \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e}; \mathsf{e}', \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e}.\mathsf{m}(\mathsf{e}'), \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}'\end{array}}$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \iota, \mathsf{H}_1, \mathsf{S}_1 \\ \mathsf{e}_2, \mathsf{H}_1, \mathsf{S}_1 \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}'\end{array}}{\begin{array}{l}\mathsf{e}_1.\mathsf{m} = \mathsf{e}_2, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}' \\ \mathsf{e}_1.\mathsf{m}(\mathsf{e}_2), \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}'\end{array}}$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \iota, \mathsf{H}_1, \mathsf{S}_1 \\ \mathsf{e}_2, \mathsf{H}_1, \mathsf{S}_1 \twoheadrightarrow \ \mathsf{v}', \mathsf{H}_2, \mathsf{S}' \\ \mathsf{H}_2(\iota)(\mathsf{m}) = \mathsf{f} \\ \mathsf{P}(\mathsf{f}) = \mathtt{function}\ \mathsf{f}(\mathtt{x} : \mathsf{t}') : \mathsf{t}''\{\ \mathtt{this} : \mathsf{t};\ \mathsf{e}\} \\ \mathsf{e}', \mathsf{H}_2, \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}'\} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}''\end{array}}{\mathsf{e}_1.\mathsf{m}(\mathsf{e}_2), \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}'}$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}, \mathsf{H}_1, \mathsf{S}_1 \\ \mathsf{e}_2, \mathsf{H}_1, \mathsf{S}_1 \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}'\end{array}}{\mathsf{e}_1; \mathsf{e}_2, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}'}$$

$$\frac{\begin{array}{l}\mathsf{e}, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}', \mathsf{H}_1, \mathsf{S}' \\ \mathsf{P}(\mathsf{f}) = \mathtt{function}\ \mathsf{f}(\mathtt{x} : \mathsf{t}') : \mathsf{t}''\{\ \mathtt{this} : \mathsf{t};\ \mathsf{e}\} \\ \iota \text{ is new in } \mathsf{H}_1 \text{ and } \mathsf{H}_2 = \mathsf{H}_1[\iota \mapsto [\![]\!]] \\ \mathsf{e}', \mathsf{H}_2, \{\mathtt{this} \mapsto \iota, \mathtt{x} \mapsto \mathsf{v}'\} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}''\end{array}}{\begin{array}{l}\mathsf{f}(\mathsf{e}), \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}' \\ \#1\end{array}}$$

$$\frac{\begin{array}{l}\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}' \\ \text{ or } (\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{v}, \mathsf{H}'', \mathsf{S}'' \text{ and } \mathsf{e}_2, \mathsf{H}'', \mathsf{S}'' \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}' \text{ and } \mathsf{v} > 0) \\ \text{ or } (\mathsf{e}_1, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ 0, \mathsf{H}'', \mathsf{S}'' \text{ and } \mathsf{e}_3, \mathsf{H}'', \mathsf{S}'' \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}')\end{array}}{\mathsf{e}_1 ?\ \mathsf{e}_2 :\ \mathsf{e}_3, \mathsf{H}, \mathsf{S} \twoheadrightarrow \ \mathsf{dv}, \mathsf{H}', \mathsf{S}'}$$

Fig. A.3. Operational Semantics - propagation of exceptions