

Towards Type Inference for JavaScript*

Christopher Anderson¹, Paola Giannini², and Sophia Drossopoulou¹

¹ Department of Computing, Imperial College London,
180 Queen's Gate, London SW7 2BZ, U.K.

² Dipartimento di Informatica, Università del Piemonte Orientale, Via Bellini 25/G,
Alessandria, Italy.

Abstract. Object-oriented scripting languages like JavaScript and Python are popular partly because of their dynamic features. These include the runtime modification of objects and classes through addition of fields or updating of methods. These features make static typing difficult and so usually dynamic typing is used. Consequently, errors such as access to non-existent members are not detected until runtime.

We first develop a formalism for an object based language, JS₀, with features from JavaScript, including dynamic addition of fields and updating of methods. We give an operational semantics and static type system for JS₀ using structural types. Our types allow objects to evolve in a controlled manner by classifying members as *definite* or *potential*.

We define a type inference algorithm for JS₀ that is sound with respect to the type system. If the type inference algorithm succeeds, then the program is typeable. Therefore, programmers can benefit from the safety offered by the type system, without the need to write explicitly types in their programs.

1 Introduction

The popularity of scripting languages stems from the flexible programming features they support. These include the runtime modification of objects through addition of fields or updating of methods. These features make static typing difficult and so usually dynamic typing is used. Consequently, errors such as access to non-existent members are not detected until runtime, or, as in JavaScript, not detected at all which can result in a web browser reporting an error when viewing a web page containing JavaScript code.

We introduce JS₀, a formalism of JavaScript[16]. JS₀ supports the standard JavaScript flexible features, e.g. functions creating objects, and dynamic addition/reassignment of fields and methods. We also introduce JS₀^T, an explicitly typed version of JS₀. Types in JS₀^T comprise object types, function types, and `Int` (the type of integers). Object types list the methods and fields present in the object, $\mu \alpha. [\mathbf{m}_1 : (\mathbf{t}_1, \psi_1) \cdots \mathbf{m}_n : (\mathbf{t}_n, \psi_n)]$. We use the μ -binder to allow a type to

* Work partly supported by EU within the FET - Global Computing initiative, project DART IST-2001-33477, MURST Cofin'02 project McTati, and MIUR Prin'04 project EOS

refer to itself. Our type system permits objects to evolve in a controlled manner by allowing members to be added to an object after it has been created. This is achieved by annotating with ψ , each member of an object type as either *potential* ‘ \circ ’ or *definite* ‘ \bullet ’.

Function types have the form, $\mathbf{t} = \mu \alpha.(\mathbf{O} \times \mathbf{t}_1 \rightarrow \mathbf{t}_2)$, where \mathbf{O} is the type of the receiver, \mathbf{t}_1 is the type of the formal parameter and \mathbf{t}_2 is the return type. As for object types, the bound variable α allows references to \mathbf{t} within \mathbf{O} , \mathbf{t}_1 , and \mathbf{t}_2 . Thus, $\mu \alpha.(- \times - \rightarrow \alpha)$ is a function that returns a value of the same type as the function itself.

A function can be used as a *global function* if its type does not make any requirements of its receiver. The type system is rich enough to allow typing of many JavaScript programs, and at the same time prevents runtime errors such as access to non-existing members of objects.

We develop a sound type inference algorithm to automatically translate JS_0 code to JS_0^T code. The algorithm uses *type variables* which represent the type of expressions. Constraints are generated between the type variables. If there is a solution to the constraints this can be used to translate code from JS_0 to JS_0^T . We define a translation between constraints and types that provides the types for the typed version of the code.

In [6] we introduced the language JS_0 and its type system. In this paper we have simplified the presentation of JS_0^T and its type system and defined a sound type inference algorithm.

This paper is organized as follows. In Section 2 we define the syntax of JS_0 and its operational semantics, and in Section 3 we give JS_0^T . Properties of the type system for JS_0^T are outlined in Section 4. In Section 5 we show type inference for JS_0 , and in Section 6 we show how to turn constraints into types. In Section 7 we compare our work with others. In Section 8 we draw conclusions and outline our future directions. The proofs and a prototype implementation can be found at <http://www.binarylord.com/work/js0/>.

2 JS_0

We have developed JS_0 a subset of JavaScript. Figure 1 gives an example JS_0 program that describes an implementation of the JavaScript Date object¹. We define functions `Date` and `addFn`. The code preceded by the comment `//Main` is the entry point into the program. Although the syntax of JS_0 requires all code to be within a function body to aid presentation we allow a main body of code and the declaration of local variables `x` and `y`. The example demonstrates the *core* JavaScript features we have included:

1. creating objects using functions (line 10 and 11),
2. implicit creation of members in objects through assignment (lines 2 and 3),
and

¹ For more information on the `Date` object see [15]. We give a simplified version and allow the adding of one date to another, with `add`.

3. acquiring methods through assignment of a function to a member (line 3).

```
1 function Date(x) {  
2   this.mSec = x;  
3   this.add = addFn;  
4   this  
5 }  
6 function addFn(x) {  
7   this.mSec = this.mSec + x.mSec; this  
8 }  
9 //Main  
10 x = new Date(1000);  
11 y = new Date(100);  
12 x.add(y);
```

Fig. 1. Untyped JS₀ Date Example

We chose these features because, (1) represents the way objects are created in JavaScript, (2) and (3) represent the way objects acquire fields and methods thus giving flexibility to the programmer. JS₀ does not include the following JavaScript features: member names as strings, functions as expressions, dynamic removal of members, automatic conversions, and delegation. We omitted the first three as we believe they are not essential in supporting flexible object-oriented programming. The last two while useful can complicate static typing and type inference. We can write the introductory examples from [15] in JS₀ assuming libraries of functions, and predefined types *e.g.* floats, strings, etc.

The syntax of JS₀ is given in Figure 2. Note that, in the syntax of JS₀ we omitted conditional expressions, which were present in [6]. Their presence does not produce conceptual difficulties regarding the type system and type inference. A program is a sequence of function declarations. In JS₀ functions may have only one formal parameter. The extension to functions with multiple parameters is trivial, whereas going to a variable number of parameters, as in JavaScript, is an interesting possible future extension.

For a program P , we use $P(f)$ as a shorthand for looking up the definition of function f in P .

2.1 Operational Semantics

We give a structural operational semantics for JS₀ 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 \rightarrow is:

$$\rightarrow : Program \rightarrow Exp \times Heap \times Stack \rightarrow (Val \cup Dev) \times Heap \times Stack$$

$P \in Program$	$::= F^*$	
$F \in FuncDecl$	$::= \text{function } f(x) \{ e \}$	
$e \in Exp$	$::= \text{var}$	locals
	f	function identifier
	$\text{new } f(e)$	object creation
	$e; e$	sequence
	$e.m(e)$	member call
	$e.m$	member select
	$f(e)$	global call
	$\text{lhs} = e$	assignment
	null	null
	n	integer
$\text{var} \in EnvVars$	$::= \text{this} \mid x$	
$\text{lhs} \in LeftHandSide$	$::= x \mid e.m$	
Identifiers		
$f \in FuncID$	$::= f \mid f' \mid \dots$	
$m \in MemberID$	$::= m \mid m' \mid \dots$	

Fig. 2. Syntax of JS₀

where:

$$\begin{aligned}
H &\in Heap = Addr \rightarrow_{\text{fin}} Obj \\
\chi &\in Stack = \{\text{this}, x\} \rightarrow Val \text{ such that } \chi(\text{this}) \in Addr \\
v &\in Val = \{\text{null}\} \cup FuncID \cup Addr \cup Int \\
dv &\in Dev = \{\text{nullPtrExc}, \text{stuckErr}\} \\
o &\in Obj = MemberID \rightarrow_{\text{fin}} Val
\end{aligned}$$

The heap maps addresses to objects, where addresses, $Addr$, are $\iota_0, \dots, \iota_n \dots$. We use \rightarrow_{fin} to indicate a finite mapping. As usual, the notation $f[x \mapsto y]$ denotes updating function f to map x to y . Thus, the meaning of heap update $H[\iota \mapsto v]$ and stack update $\chi[x \mapsto v]$ is clear. 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 from member identifiers to values. With $\langle m_1 : v_1 \dots m_n : v_n \rangle$ we denote the object mapping m_i to v_i for $i \in 1 \dots n$.

A full description of the rules is given in [6]. In JavaScript access to non-existent members result in an undefined value not a runtime error². This may cause errors later on in the code. We consider accesses to non-existent members a runtime error and our type system prevents them. Below we give two of the

² For some interesting insights into issues surrounding JavaScript's treatment of undefined members see [24].

more interesting rules, (*memAdd*) for adding/updating members and (*memCall*) for calling methods:

$$\begin{array}{c}
\frac{e_1, H, \chi \rightarrow \iota, H_1, \chi_1 \quad e_2, H_1, \chi_1 \rightarrow v, H_2, \chi' \quad H' = H_2[\iota \mapsto H_2(\iota)[m \mapsto v]]}{e_1.m = e_2, H, \chi \rightarrow v, H', \chi'} \text{ (memAdd)} \\
\frac{e_1, H, \chi \rightarrow \iota, H_1, \chi_1 \quad e_2, H_1, \chi_1 \rightarrow v', H_2, \chi' \quad H_2(\iota)(m) = f \quad P(f) = \text{function } f(x) \{e'\} \quad \chi_2 = \{\text{this} \mapsto \iota, x \mapsto v'\} \quad e', H_2, \chi_2 \rightarrow v, H', \chi''}{e_1.m(e_2), H, \chi \rightarrow v, H', \chi'} \text{ (memCall)}
\end{array}$$

In rule (*memAdd*) we express how objects obtain new members. We first evaluate the receiver, then the right hand side. Using heap update we add/update member m in the receiver. Returning to the example in Figure 1, executing `this.mSec = x` from `Date` with $\chi_0(\text{this}) = \iota_0$, $\chi_0(x) = 1000$, $H_0(\iota_0) = \langle\langle\rangle\rangle$, will produce H_1 with $H_1(\iota_0) = \langle\langle \text{mSec} : 1000 \rangle\rangle$

In rule (*memCall*) 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^3 . 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.

For example, executing the code in Figure 1 in the presence of an empty heap, H_0 and χ_0 , mapping x and y to `null` will result in stack $\chi_1(x) = \iota_0$, $\chi_1(y) = \iota_1$ and updated heap H_1 , $H_1(\iota_0) = \langle\langle \text{mSec} : 1100, \text{add} : \text{addFn} \rangle\rangle$, $H_1(\iota_1) = \langle\langle \text{mSec} : 100, \text{add} : \text{addFn} \rangle\rangle$. For demonstration purposes the stack contains an extra variable y although the definition of stack allows only `this` and x .

Note that member `add` of both ι_0 and ι_1 has value `addFn`. This indicates that it is an alias of function `addFn`, which is invoked when `x.add(y)` is executed.

3 A Type System for JS₀

In this section we introduce JS₀^T a typed version of JS₀. Figure 3 shows the parts of JS₀^T that differ from JS₀ along with the definitions of types. Observe that functions are now annotated with a function type G .

Types t_1, \dots, t_n , comprise object types, function types, or `Int` (the type of integers). Object types list the methods and fields present in the object. We use the μ -binder to allow a type to refer to itself. So $\mu \alpha.M$ where $M = [m_1 : (t_1, \psi_1) \cdots m_n : (t_n, \psi_n)]$, is the type of an object with members m_1, \dots, m_n of type t_1, \dots, t_n , respectively. Figure 4 gives a JS₀^T version of the `Date` example from Figure 1. We use t_1 for type $[mSec : (\text{Int}, \circ), \text{add} : ((t_2 \times t_2 \rightarrow t_2), \circ)]$ and t_2 for type $\mu \alpha.[mSec : (\text{Int}, \bullet), \text{add} : ((\alpha \times \alpha \rightarrow \alpha), \bullet)]$. To aid the presentation we allow local variable type declarations on lines 10 and 11. These are not part of the syntax of JS₀^T, where type declarations are only allowed for the

³ For clearness of presentation we omit P from the reduction rules.

Syntax	
$P \in Program$	$::= F^*$
$F \in FuncDecl$	$::= \text{function } f(x) : G \{ e \}$
Types	
$t \in Type$	$::= O \mid G \mid Int$
$tp \in PreType$	$::= \alpha \mid t$
$O \in ObjType$	$::= \mu \alpha. M \mid M$
$G \in FuncType$	$::= \mu \alpha. R \mid R$
$M \in ObjMembers$	$::= [(m : tm)^*]$
$tm \in MemberType$	$::= (tp, \psi)$
$R \in FuncRow$	$::= (O \times tp \rightarrow tp)$
$\psi \in Annotation$	$::= \circ \mid \bullet$
$\alpha \in ObjVar$	$::= \alpha \mid \alpha' \mid \alpha'' \dots$

Fig. 3. Syntax of JS_0^T

parameter of a method and `this` and are implicitly given in the function type for a function.

```

1 function Date(x):(t1 × Int → t2) {
2   this.mSec = x;
3   this.add = addFn;
4   this
5 }
6 function addFn(x):(t2 × t2 → t2) {
7   this.mSec = this.mSec + x.mSec; this;
8 }
9 //Main
10 t2 x = new Date(1000);
11 t2 y = new Date(100);
12 x.add(y);
```

Fig. 4. Typed JS_0 Date Example.

Our type system permits objects to evolve in a controlled manner by allowing members to be added to an object after it has been created. This is achieved by annotating each member of an object type as either *potential* ‘ \circ ’ or *definite* ‘ \bullet ’ e.g. $mSec : (Int, \circ)$ in t_1 and $mSec : (Int, \bullet)$ in t_2 . When a potential member

is assigned to, it becomes definite, replacing \circ with \bullet . To keep the type system manageable we only track assignments to variables (formal parameters and `this`) within the scope of a function. In a well-typed program potential members may not be accessed until they have been assigned to.

Function types, $(\mathbf{O} \times \mathbf{t}_1 \rightarrow \mathbf{t}_2)$ or $\mu \alpha.(\mathbf{O} \times \mathbf{t}_1 \rightarrow \mathbf{t}_2)$, list the type of the receiver, \mathbf{O} , which is an object type, the type of the parameter, \mathbf{t}_1 , and the type of the return value of the function, \mathbf{t}_2 . As for object types the μ -binder allows a function type to refer to itself, thus $\mu \alpha.(- \times - \rightarrow \alpha)$ is a function that returns a function with its type.

If the type of \mathbf{m} is an object type, or Int , the member represents a field. If the type of \mathbf{m} is a function type, then \mathbf{m} represents a method. In case the type of the \mathbf{m} is α then if α is bound in an objects type the member is a field, whereas if it is bound in a function type it is a method. An object type is well-formed if it is closed and contains unique member definitions that are themselves well-formed. A function type $\mathbf{G} = \mu \alpha.\mathbf{R}$ (or $\mathbf{G} = \mathbf{R}$) is well-formed, $\vdash \mathbf{G} \diamond$, if the receiver, parameter and return types of $\mathbf{G}[\alpha/\mathbf{R}]$ (or \mathbf{R}) are well-formed.

For a well-formed object type \mathbf{O} , define $\mathbf{O}(\mathbf{m})$, which selects the annotated type of the member \mathbf{m} in \mathbf{O} (if it is defined) by first defining selection from $\mathbf{O} = [\mathbf{m}_1 : (\mathbf{t}_1, \psi_1) \cdots \mathbf{m}_n : (\mathbf{t}_n, \psi_n)]$ as

$$\mathbf{O}(\mathbf{m}) = \begin{cases} (\mathbf{t}_i, \psi_i) & \text{if } \mathbf{m} = \mathbf{m}_i \text{ for some } i, 1 \leq i \leq n \\ \text{Udf} & \text{otherwise} \end{cases}$$

and then if $\mathbf{O} = \mu \alpha.\mathbf{M}$,

$$\mathbf{O}(\mathbf{m}) = \mathbf{M}[\alpha/\mathbf{O}](\mathbf{m})$$

That is, the type is closed by substituting occurrences of α with the enclosing type. Therefore, if \mathbf{O} is well-formed, then also $\mathbf{O}(\mathbf{m})$ is well-formed.

With $\mathbf{O}[\mathbf{m} \mapsto (\mathbf{t}, \psi)]$ we denote the *updating of the member \mathbf{m} to type \mathbf{t} with annotation ψ in \mathbf{O}* . Note that, if \mathbf{O} and \mathbf{t} are well-formed, then $\mathbf{O}[\mathbf{m} \mapsto (\mathbf{t}, \psi)]$ is well-formed.

Congruence and Subtyping *Congruence* between types is defined in Figure 5. With $\mathbf{t}_1[\alpha/\mathbf{t}_2]$, we denote the substitution of the free occurrences of α in \mathbf{t}_1 with \mathbf{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 $\mathbf{t} \leq \mathbf{t}'$, defined in Figure 6, means that an object or function of type \mathbf{t} can be used whenever one of type \mathbf{t}' is required. For object types we have subtyping in width. If $\mathbf{O} \leq \mathbf{O}'$, then all definite members of \mathbf{O}' must be present and congruent with those in \mathbf{O} , and all potential members of \mathbf{O}' must be present as potential or definite members of \mathbf{O} with congruent types. This condition is needed to insure that the addition of a new member to an object does not break compatibility.

Returning to the example in Figure 4 we see that \mathbf{t}_2 is a subtype of \mathbf{t}_1 because all members of \mathbf{t}_1 are also members of \mathbf{t}_2 , and have congruent types; furthermore, all members of \mathbf{t}_2 are definite.

Reflexivity $\frac{}{t \equiv t}$	Unfolding $\frac{}{\mu \alpha.M \equiv M[\alpha/\mu \alpha.M]} \\ \mu \alpha.R \equiv R[\alpha/\mu \alpha.R]$	Transitivity $\frac{t_1 \equiv t_2 \quad t_2 \equiv t_3}{t_1 \equiv t_3}$
Alpha – conversion		
$\frac{\alpha' \notin \mathcal{FV}(M)}{\mu \alpha.M \equiv \mu \alpha'.M[\alpha/\alpha']}$	$\frac{\alpha' \notin \mathcal{FV}(R)}{\mu \alpha.R \equiv \mu \alpha'.R[\alpha/\alpha']}$	
Reordering $\frac{\forall m : M(m) \equiv M'(m)}{M \equiv M'}$	Functions $\frac{M \equiv M' \quad t_1 \equiv t'_1 \quad t_2 \equiv t'_2}{(M \times t_1 \rightarrow t_2) \equiv (M' \times t'_1 \rightarrow t'_2)}$	Members $\frac{t \equiv t'}{(t, \psi) \equiv (t', \psi)}$

Fig. 5. Congruence for types

For function types subtyping coincides with congruence. In future versions of this work we may relax this restriction and allow contravariance of the receiver and parameter type and covariance of the return type. However, since type inference was our main aim, we started with the reduced system. Given types t and t' it is decidable whether $t \leq t'$ or not.

$\frac{\psi' = \bullet \implies \psi = \bullet}{\psi \leq \psi'}$	$\frac{t \equiv t' \quad \psi \leq \psi'}{(t, \psi) \leq (t', \psi')}$	$\frac{t \equiv t'}{t \leq t'}$
$\frac{\forall m : O'(m) = (t', \psi') \implies (O(m) = (t, \psi) \wedge (t, \psi) \leq (t', \psi'))}{O \leq O'}$		

Fig. 6. Subtyping

3.1 Typing Expressions

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

$$P, \Gamma \vdash e : t \parallel \Gamma'$$

The environment, $\Gamma = \{\mathbf{this} : O, x : t\}$, maps the receiver, \mathbf{this} , to a well-formed object type, and the formal parameter, x , to a well-formed type. The environment on the right hand side of the judgement, Γ' , reflects the changes to the type of the receiver or parameter while typing the expression. The only possible difference between Γ and Γ' is that some members that are annotated with \circ in Γ are annotated with \bullet in Γ' . With $\Gamma[\mathbf{var} \mapsto t]$ we denote the *updating of var to type t in Γ* .

Consider the typing rules of Figure 7. Rules (*var*), (*func*), (*const*), and (*seq*) are straightforward. Note that `null` may have any object type.

In rule (*memAcc*) the expression e must be of an object type in which the member m is definite, i.e. with annotation \bullet .

We use the notation $G(\mathbf{this})$, $G(x)$, and $G(\mathbf{ret})$, to denote the types of the receiver, parameter and return value of G . As for member selection, we define for $G = (O \times t_1 \rightarrow t_2) : G(\mathbf{this}) = O \quad G(x) = t_1 \quad G(\mathbf{ret}) = t_2$ and for $G = \mu \alpha.R$, we define $G(z) = (R[\alpha/G])(z)$ where $z \in \{x, \mathbf{this}, \mathbf{ret}\}$.

Rule (*methCall*) checks that the type of the receiver is an object type in which the member m has a definite function type. Moreover, the type of the receiver and actual parameter must be subtypes of the declared type of the receiver and formal parameter.

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 (*assignAdd*) in Γ'' we ensure that member m (of \mathbf{this} or the formal parameter) is definite. From this point onwards, member m of \mathbf{var} may be accessed. For example, consider the expression $x.m_2 = x$ in the environment Γ , where $\Gamma(x)$ has type $t = \mu \alpha.[m_1 : (\text{Int}, \bullet), m_2 : (\alpha, \circ)]$. The expression is well-typed in Γ and we have $P, \Gamma \vdash x.m_2 = x : t \parallel \Gamma'$ where Γ' maps \mathbf{this} to $\Gamma(\mathbf{this})$ and x to $[m_1 : (\text{Int}, \bullet), m_2 : (t, \bullet)]$. This reflects the updating of member m_2 . Any aliases to \mathbf{this} or the formal parameter will not *see* the update of a member. This would require dataflow analysis techniques and is beyond the scope of this work. The fact that the type system requires a member to be *known* (either as potential or definite) for an assignment to succeed is not a limitation. The process of type inference will find all members for a type with their appropriate type and annotation.

Rule (*assignUpd*) is used when the assignment is to a definite member m . In this case we just check that the type of the expression on the right hand side is a subtype of the type of the member m .

A program P is *well-formed* if all the function declarations in P are well-typed. Figure 7 gives the definition.

Typing Expressions

$$\begin{array}{c}
\frac{}{P, \Gamma \vdash \text{this} : \Gamma(\text{this}) \parallel \Gamma} \text{ (var)} \qquad \frac{P(f) = \text{function } f(x) : G\dots}{P, \Gamma \vdash f : G \parallel \Gamma} \text{ (func)} \\
\frac{}{P, \Gamma \vdash \text{null} : O \parallel \Gamma} \text{ (const)} \qquad \frac{P, \Gamma \vdash e_1 : t \parallel \Gamma' \quad P, \Gamma' \vdash e_2 : t' \parallel \Gamma''}{P, \Gamma \vdash e_1; e_2 : t' \parallel \Gamma''} \text{ (seq)} \\
\frac{P, \Gamma \vdash e : O \parallel \Gamma' \quad O(m) = (t', \bullet)}{P, \Gamma \vdash e.m : t' \parallel \Gamma'} \text{ (memAcc)} \qquad \frac{P, \Gamma \vdash e : t \parallel \Gamma' \quad t \leq \Gamma'(x)}{P, \Gamma \vdash x = e : t \parallel \Gamma'} \text{ (varAss)} \\
\frac{P, \Gamma \vdash e_1 : O \parallel \Gamma' \quad O(m) = (G, \bullet) \quad P, \Gamma' \vdash e_2 : t' \parallel \Gamma'' \quad t' \leq G(x) \quad O \leq G(\text{this})}{P, \Gamma \vdash e_1.m(e_2) : G(\text{ret}) \parallel \Gamma''} \text{ (methCall)} \qquad \frac{P, \Gamma \vdash e : t \parallel \Gamma' \quad P(f) = \text{function } f(x) : G\dots \quad t \leq G(x) \quad \{t' \mid (G(\text{this}))(m) = (t', \bullet)\} = \emptyset}{P, \Gamma \vdash \text{new } f(e) : G(\text{ret}) \parallel \Gamma' \quad P, \Gamma \vdash f(e) : G(\text{ret}) \parallel \Gamma'} \text{ (call)} \\
\frac{P, \Gamma \vdash e_2 : t \parallel \Gamma' \quad \Gamma'(\text{var}) = O \quad O(m) = (t'', \psi) \quad t \leq t'' \quad \Gamma'' = \Gamma'[\text{var} \mapsto O[m \mapsto (t'', \bullet)]]}{P, \Gamma \vdash \text{var}.m = e_2 : t \parallel \Gamma''} \text{ (assignAdd)} \qquad \frac{P, \Gamma \vdash e_1 : O \parallel \Gamma' \quad P, \Gamma' \vdash e_2 : t \parallel \Gamma'' \quad O(m) = (t'', \bullet) \quad t \leq t''}{P, \Gamma \vdash e_1.m = e_2 : t \parallel \Gamma''} \text{ (assignUpd)}
\end{array}$$

Well – formed Programs

$$\frac{\forall f : \quad P(f) = \text{function } f(x) : G \{e\} \wedge \vdash G \diamond \quad \Rightarrow \quad P, \{ \text{this} : G(\text{this}), x : G(x) \} \vdash e : t \parallel \Gamma'' \wedge t \leq G(\text{ret})}{\vdash P \diamond}$$

Fig. 7. Type Rules for Expressions in JS_0^\top

4 Formal Properties of the Type System

In this section we give the relevant definitions and the statement that asserts that our type system is sound w.r.t. to the operational semantics given in Section

2.1. 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 between values and well-formed types should have.

Definition 1. *Given a heap, H , and a program, P , we say that $A \subseteq (Val \times Type)$ is an agreement relation if the following conditions are satisfied:*

- if $(\text{null}, t) \in A$, then $t = O$ for some well-formed O ,
- if $(n, t) \in A$, then $t = \text{Int}$,
- if $(f, t) \in A$, then $P(f) = \text{function } f(x) : G \text{ and } G \equiv t$,
- if $(l, t) \in A$, then
 - $t = O$ for some well-formed O , $H(l) = \ll m_1 : v_1 \dots m_p : v_p \gg$, and
 - $O(m) = (t', \bullet) \implies m = m_i$ for some $i, i \in 1..p$, and $(v_i, t') \in A$
 - $O(m) = (t', \circ)$ and $m = m_i$ for some $i, i \in 1..p$, $\implies (v_i, t') \in A$

If A and A' are agreement relations, then $A \cup A'$ is also an agreement relation. Therefore, the union of all agreement relations defines a relation between values and types, which determines when a value has a given type.

Definition 2. Value v is compatible with type t in H

$$P, H \vdash v \triangleleft t$$

if $(v, t) \in A$ for some agreement relation A on H and P .

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

Lemma 1. *If $t \leq t'$ and $P, H \vdash v \triangleleft t$ then $P, H \vdash v \triangleleft t'$.*

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

Definition 3. $P, \Gamma \vdash H, \chi \diamond$ holds if $P, H \vdash \chi(\text{this}) \triangleleft \Gamma(\text{this})$ and $P, H \vdash \chi(x) \triangleleft \Gamma(x)$.

We can now state the Soundness Theorem. The theorem asserts that if an expression is well-typed,

$$P, \Gamma \vdash e : t \parallel \Gamma'$$

then the evaluation of the expression starting in a heap and stack that are compatible with Γ will not get stuck. That is, the result of the evaluation is either a value compatible with type t , or it is a `nullPtrExc` exception. In particular, it is not a `stuckErr` error. Moreover, the stack and heap produced are compatible with the final environment Γ' .

Theorem 1. *[Type Soundness] For a well-formed program P , environment Γ , and expression e , such that:*

$$P, \Gamma \vdash e : t \parallel \Gamma'$$

If $P, \Gamma \vdash H, \chi \diamond$ and $e, H, \chi \rightarrow w, H', \chi'$, then either

- $w = \text{nullPtrExc}$, or
- $w = v$, $P, H' \vdash v \triangleleft t$, and $P, \Gamma' \vdash H', \chi' \diamond$.

5 Type Inference

We show how type inference for JS_0 can be expressed as a finite system of constraints between type variables. Type variables are used to represent the type of an expression. From a JS_0 program, we can generate a set of type variables with constraints between them. Constraints represent the relationships we expect between types in the program. For example, that the actual parameter to a function call should be a subtype of the formal parameter.

If the constraints have a solution we say that they are satisfiable. A solution can be used to translate a JS_0 program into an equivalent JS_0^T program. This involves annotating the JS_0 program with type declarations. We show that the annotated program is well-typed.

5.1 Type Variables

As in [20,3,21,18], we use type variables to express the - yet unknown - types of expressions. Thus, $\llbracket \text{new Date}(1000) \rrbracket$ expresses the type of `new Date(1000)`.

Because the types of `this` and `x` differ for different occurrences in the same method body, we use labels to distinguish them, for example, $\llbracket \text{this}_1 \rrbracket$, $\llbracket x_2 \rrbracket$, $\llbracket \text{this}_3 \rrbracket$, *etc.*. Labeled type variables⁴ $\llbracket \text{this}_f \rrbracket$ and $\llbracket x_f \rrbracket$ represent the type of `this` and `x` at the beginning of the function `f`, and $\llbracket \text{ret}_f \rrbracket$ represents the return type of the function.

We generate a new label for each method call; this label is used to generate three type variables. These variables denote the type of the receiver, parameter and return type of the method. For example, for `x.add(y)` we could use label `5` which would generate $\llbracket \text{call_this}_5 \rrbracket$, $\llbracket \text{call}_x_5 \rrbracket$, and $\llbracket \text{call_ret}_5 \rrbracket$. Note that these type variables depend on the label but not on the name of the method.⁵ Figure 8 gives the syntax of labeled expressions.

Figure 9 defines type variables. Type variables can be used to describe function types, *e.g.* $(\tau \times \tau' \rightarrow \tau'')$, or object types, *e.g.* $\llbracket m:(\tau, \psi) \rrbracket$ with the obvious meaning.

5.2 Constraints and Solutions

A solution, S , is a mapping from type variables to types. For the `Date` example, let t_2 be $\mu \alpha. \llbracket \text{mSec} : (\text{Int}, \bullet), \text{add} : ((\alpha \times \alpha \rightarrow \alpha), \bullet) \rrbracket$, S_0 represents part of a solution, as follows:

$$\begin{aligned} S_0(\llbracket \text{this_Date} \rrbracket) &= \llbracket \text{mSec} : (\text{Int}, \circ), \text{add} : ((t_2 \times t_2 \rightarrow t_2), \circ) \rrbracket \\ S_0(\llbracket \text{this}_1 \rrbracket) &= \llbracket \text{mSec} : (\text{Int}, \bullet), \text{add} : ((t_2 \times t_2 \rightarrow t_2), \circ) \rrbracket \\ S_0(\llbracket \text{ret_Date} \rrbracket) &= t_2 \quad S_0(\llbracket x_Date \rrbracket) = \text{Int} \\ S_0(\llbracket \text{this_Date.mSec} \rrbracket) &= \text{Int} \quad S_0(\llbracket \text{this}_5 \rrbracket) = \llbracket \text{mSec} : (\text{Int}, \bullet) \rrbracket \end{aligned}$$

⁴ In [3] Agesen et al. use a similar labeling, $\llbracket e \rrbracket_\tau$, to indicate who the sender, τ , of a method call is.

⁵ It is possible to optimize the creation of new variables at the call site, for example by sharing some of them. Refer to Section 7 where we discuss [25] which shows possible optimizations.

$e \in LabExp$	$::= \text{void} \mid f \mid \text{new } f(e) \mid e; e \mid e.m(e) \mid$
	$e.m \mid f(e) \mid \llbracket \text{hs} = e \mid \text{null} \mid n \mid \text{ie} \rrbracket$
$\text{void} \in LabEnvVars$	$::= \text{this}_l \mid x.l$
$\llbracket \text{hs} \in LeftSide \rrbracket$	$::= x.l \mid e.m$
$\text{ie} \in InferExp$	$::= \text{ret } f \mid \text{call_this}_l \mid \text{call_x}_l \mid \text{call_ret}_l$
$l \in Lab$	$::= 1 \mid 2 \mid \dots \mid f \mid f' \mid \dots$

Fig. 8. Syntax of Labeled Expressions

Constraints between type variables express the relationship between the types of expressions, i.e. which members a type must have, how the members of two types may differ and whether a type has any definite members. The syntax of constraints is given in Figure 9. There are three kinds of constraint: $\tau \leq \rho$, $\tau \triangleleft \tau$, and τ° . We use c to range over constraints and C for a set of constraints.

Figure 10, rule (*solSat*), defines that S satisfies a set of constraints, $S \vdash C$, if it satisfies each constraint. We now discuss each kind of constraint and how it is satisfied by a solution.

- $\tau \leq \rho$ - requires a type variable to be a subtype of ρ : Thus, $\tau \leq \text{Int}$ requires τ to be Int , *c.f.* rule (*solInt*); while $\tau \leq \tau'$ requires τ to be a subtype of τ' , *c.f.* (*solSub*); while $\tau \leq (\tau_1 \times \tau_2 \rightarrow \tau_3)$ requires τ to be the function type composed from τ_1 , τ_2 and τ_3 , *c.f.* (*solSubFunc*); finally, $\tau \leq [m : (\tau', \psi)]$ requires τ to have a member m of type τ' with annotation at least ψ , *c.f.* (*solMemChange*).

Thus, $S_0 \vdash \llbracket \text{this}_1 \rrbracket \leq \llbracket \text{this_Date} \rrbracket$, and $S_0 \vdash \llbracket \text{this}_1 \rrbracket \leq \llbracket \text{this}_5 \rrbracket$,

$S_0 \vdash \llbracket \text{this_Date} \rrbracket \leq [mSec : (\llbracket \text{this_Date.mSec} \rrbracket, o)]$, but

$S_0 \not\vdash \llbracket \text{this_Date} \rrbracket \leq [mSec : (\llbracket \text{this_Date.mSec} \rrbracket, \bullet)]$.

- $\tau \triangleleft_m \tau'$ - requires τ and τ' to have the same members with the same types, but member m can be potential in τ' but must be definite in τ , *c.f.* rule (*solMemChange*).

For example, $S_0 \vdash \llbracket \text{this}_1 \rrbracket \triangleleft_{mSec} \llbracket \text{this_Date} \rrbracket$, while

$S_0 \not\vdash \llbracket \text{this_Date} \rrbracket \triangleleft_{mSec} \llbracket \text{this}_1 \rrbracket$. Also $S_0 \not\vdash \llbracket \text{this}_1 \rrbracket \triangleleft_{mSec} \llbracket \text{this}_5 \rrbracket$. Note,

however, that $S_0 \vdash \llbracket \text{this}_1 \rrbracket \leq \llbracket \text{this}_5 \rrbracket$ – this should clarify the difference between the two kinds of constraint.

- τ° - requires τ to have no definite members, *c.f.* rule (*solNoDefs*). This is needed for constructors and global functions whose receiver must have no definite members. For example, $S_0 \vdash \llbracket \text{this_Date} \rrbracket^\circ$.

5.3 Constraint Generation

Constraint generation for a JS_0 program produces a set of constraints between type variables, and a labeled version of the original expression, e . A

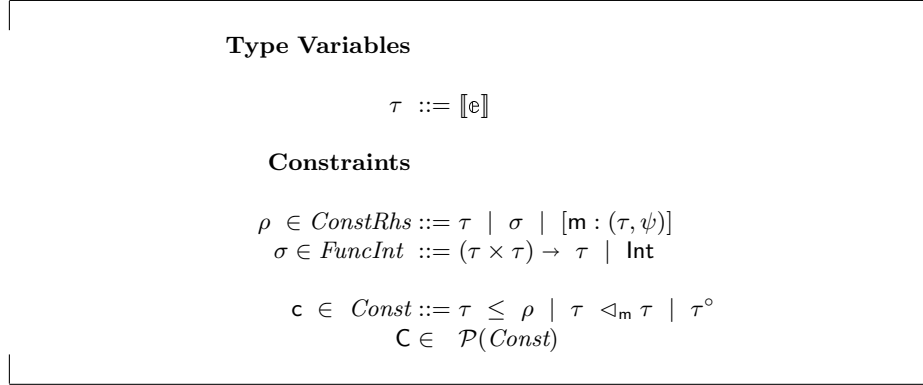


Fig. 9. Syntax of Type Variables and Constraints

pre-environment, $\gamma = \{\mathbf{this} : \mathbf{l}, \mathbf{x} : \mathbf{l}', \mathbf{lab} : \mathbf{L}\}$, keeps track of the current labeling of **this** and **x** along with the set of labels used so far, stored in the set **L**. Constraint generation for an expression **e** in the context of a pre-environment, γ , has the form:

$$\gamma \vdash e : e \parallel \gamma' \parallel \mathbf{C}$$

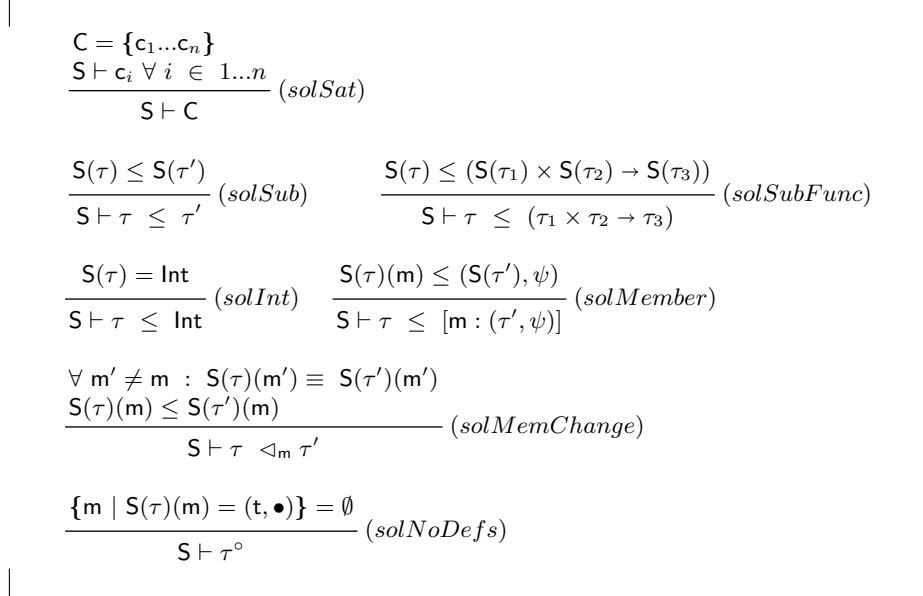


Fig. 10. Solution Satisfaction

where γ' reflects the changes to the labeling of **this**, **x** and **lab** while generating constraints. The constraints generated for an expression consist of the union of the constraints for each subexpression augmented by local constraints.

In (*var*) we generate a labeled expression for **this** and **x** by looking in the pre-environment for the current label. No constraints are generated.

In (*funcId*) we require **f** to have a function type derived from the type of the receiver, parameter and return value of the function. The type variables come from the initial labeled **this** and **x** and the labeled return variable **ret.f**. For example, function identifier **addFn** produces constraint:

$$\llbracket \text{addFn} \rrbracket \leq (\llbracket \text{this.addFn} \rrbracket \times \llbracket \text{x.addFn} \rrbracket \rightarrow \llbracket \text{ret.addFn} \rrbracket)$$

In (*assignAdd*) we use **var** for **this** or **x**, and we model the change of member **m** of **var** to definite. **var.l** and **var.l'** represent the type of **var** *before* and *after* the update, where **l'** is fresh. Constraint $\llbracket \text{var.l} \rrbracket \leq [m : (\llbracket \text{var.l.m} \rrbracket, \circ)]$ requires **var** to have member **m** with annotation at least \circ before the update, while $\llbracket \text{var.l}' \rrbracket \leq [m : (\llbracket \text{var.l}' \rrbracket, \bullet)]$ requires **var** to have member **m** with annotation definite after the update⁶. The constraint $\llbracket \text{var.l}' \rrbracket \triangleleft_m \llbracket \text{var.l} \rrbracket$ requires that only member **m** is affected by the assignment. The remaining constraints require that the type of member **m**, $\llbracket \text{var.l}' \rrbracket$, and the overall expression have the type of the right hand side of the assignment. For example, **this.add = addFn** in a pre-environment $\gamma_2 = \{\text{this} : 1, \text{lab} : L, \dots\}$ where $2 \notin L$, generates the constraints: $\llbracket \text{this}_1 \rrbracket \leq [\text{add} : (\llbracket \text{this}_1.\text{add} \rrbracket, \circ)]$, $\llbracket \text{this}_2 \rrbracket \leq [\text{add} : (\llbracket \text{this}_2.\text{add} \rrbracket, \bullet)]$, $\llbracket \text{this}_2 \rrbracket \triangleleft_{\text{add}} \llbracket \text{this}_1 \rrbracket$, $\llbracket \text{addFn} \rrbracket \leq \llbracket \text{this}_2.\text{add} \rrbracket$, $\llbracket \text{addFn} \rrbracket \leq \llbracket \text{this}_2.\text{add} = \text{addFn} \rrbracket$ and the post-environment $\gamma_2[\text{this} \mapsto 2, \text{lab} \mapsto L \cup \{2\}]$.

In (*new*) a function is used to create an object. The constraint $\llbracket \text{this.f} \rrbracket^\circ$ requires the initial **this** for **f** to have no definite members. The constraint $\llbracket \text{e} \rrbracket \leq \llbracket \text{x.f} \rrbracket$ requires the actual parameter to have a subtype of the formal parameter, where **x.f** is the type of the formal parameter at the beginning of the function body. The constraint $\llbracket \text{ret.f} \rrbracket \leq \llbracket \text{new f(e)} \rrbracket$ requires the return type of the function to be a subtype of the overall type of the **new** expression. For example, **new Date(1000)** generates constraints: $\llbracket \text{this.Date} \rrbracket^\circ, \llbracket 1000 \rrbracket \leq \llbracket \text{x.Date} \rrbracket, \llbracket \text{ret.Date} \rrbracket \leq \llbracket \text{new Date}(1000) \rrbracket$ The rule for global function (*funcCall*) is similar in structure to that for (*new*).

For member access, (*memAcc*), and for assignment where the receiver is not **this** or **x**, (*assignUpd*), the receiver must have the definite member. For example, **x.mSec**, in a $\gamma_1 = \{\text{x} : 2, \dots\}$, generates constraint: $\llbracket \text{x.2} \rrbracket \leq [m\text{Sec} : (\llbracket \text{x.2.mSec} \rrbracket, \bullet)]$

For method call, (*methCall*), we consider the label characterizing the occurrence of the call. For a call with label **l** we require the receiver to have a definite member, **m**, with function type $\llbracket \text{call.this.l} \rrbracket \times \llbracket \text{call.x.l} \rrbracket \rightarrow \llbracket \text{call.ret.l} \rrbracket$, as expressed through the con-

⁶ Using $\llbracket \text{var.l} \rrbracket \leq [m : (\llbracket \text{e} \rrbracket, -)]$ instead of $\llbracket \text{var.l} \rrbracket \leq [m : (\llbracket \text{var.l.m} \rrbracket, -)]$ would have been too restrictive. Namely, a solution would require the type of **m** to be the same as the type of $\llbracket \text{e} \rrbracket$ rather than a supertype.

straint $\llbracket e_1 \rrbracket \leq [m : (\llbracket \text{call_this_1} \rrbracket \times \llbracket \text{call_x_1} \rrbracket \rightarrow \llbracket \text{call_ret_1} \rrbracket, \bullet)]^7$. This will ensure that a solution to the constraints will give a type to the member, that is the least upper bound of all the receivers, parameters and return types at the call sites. For example, $x.\text{add}(y)$ in a pre-environment $\gamma_3 = \{x : \text{Main}, y : \text{Main}, \text{lab} : L, \dots\}$ where $5 \notin L$, generates constraints:

$\llbracket x_{\text{Main}} \rrbracket \leq [\text{add} : (\llbracket x_{\text{Main.add}} \rrbracket, \bullet)]$, $\llbracket x_{\text{Main.add}} \rrbracket \leq (\llbracket \text{call_this_5} \rrbracket \times \llbracket \text{call_x_5} \rrbracket \rightarrow \llbracket \text{call_ret_5} \rrbracket)$, $\llbracket x_{\text{Main}} \rrbracket \leq \llbracket \text{call_this_5} \rrbracket$, $\llbracket y_{\text{Main}} \rrbracket \leq \llbracket \text{call_x_5} \rrbracket$, $\llbracket \text{call_ret_5} \rrbracket \leq \llbracket x_{\text{Main.add}}(y_{\text{Main}}) \rrbracket$ and the post-environment $\gamma_3[\text{lab} \mapsto L \cup \{5\}]$.

For programs, (*Prog*), we collect the constraints generated for each function with a pre-environment mapping **this** and **x** to their respective initial versions and **lab** to the given set of labels.

5.4 Soundness of the Constraints

We now show that the constraints are sound with respect to the type system. Given a solution, S , and pre-environment, γ , we can generate an environment, Γ , as follows: $\Gamma_{\text{gen}}(\gamma, S) = \{\text{this} \mapsto S(\text{this}.\gamma(\text{this})), x \mapsto S(x.\gamma(x))\}$.

Theorem 2 guarantees soundness of the constraints at expression level: Given an expression and its constraints, if there is a solution then the type given by the type system is a subtype of that given in the solution. The environments used for type checking are those produced by Γ_{gen} with pre-environments γ and γ' .

Theorem 2. *If $\gamma \vdash e : e \parallel \gamma' \parallel C$ and $S \vdash C$ and $\Gamma = \Gamma_{\text{gen}}(\gamma, S)$ and $\Gamma' = \Gamma_{\text{gen}}(\gamma', S)$ then $P, \Gamma \vdash e : t \parallel \Gamma'$ and $t \leq S(\llbracket e \rrbracket)$.*

Theorem 3 states soundness of the constraints at the program level. Given a program and its constraints, if there is a solution we can use it to generate a well-typed version of the program. Given a JS_0 program and a solution, function $\mathcal{T}(P, S)$ generates the corresponding typed JS_0^T program, by using the solution to find the type of the formal parameter, receiver and return type of all the functions and removing the labeling.

Theorem 3. *If $\vdash P : C$ and $S \vdash C$ then $\vdash \mathcal{T}(P, S) \diamond$*

6 From Constraints to Solutions

We now discuss how constraints can be closed to make explicit a solution and how to check that constraints are well-formed. We show how a well-formed set of constraints can be used to generate a solution.

⁷ Constraint $\llbracket e_1 \rrbracket \leq [m : (\llbracket e_1 \rrbracket \times \llbracket e_2 \rrbracket \rightarrow \llbracket e_1.m(e_2) \rrbracket, \bullet)]$ would have been too restrictive. Namely, it would require all the receivers of the method to have the same type.

Constraint Generation for Expressions

$$\begin{array}{c}
\frac{}{\gamma \vdash \text{null} : \text{null} \parallel \gamma \parallel \emptyset} \\
\gamma \vdash \text{n} : \text{n} \parallel \gamma \parallel \{ \llbracket \text{n} \rrbracket \leq \text{Int} \} \\
\gamma \vdash \text{this} : \text{this} \cdot \gamma(\text{this}) \parallel \gamma \parallel \emptyset \\
\gamma \vdash \text{x} : \text{x} \cdot \gamma(\text{x}) \parallel \gamma \parallel \emptyset \\
\\
\gamma \vdash \text{e} : \text{e} \parallel \gamma'' \parallel \text{C}' \\
\gamma''(\text{var}) = 1 \\
1' \notin \gamma''(\text{lab}) \\
\gamma' = \gamma''[\text{var} \mapsto 1', \text{lab} \mapsto (\gamma''(\text{lab}) \cup \{1'\})] \\
\text{C} = \{ \llbracket \text{var.l} \rrbracket \leq [m : (\llbracket \text{var.l.m} \rrbracket, \circ)], \\
\llbracket \text{var.l}' \rrbracket \leq [m : (\llbracket \text{var.l'.m} \rrbracket, \bullet)], \\
\llbracket \text{var.l}' \rrbracket \triangleleft_m \llbracket \text{var.l} \rrbracket, \llbracket \text{e} \rrbracket \leq \llbracket \text{var.l'.m} \rrbracket, \llbracket \text{e} \rrbracket \leq \llbracket \text{var.l'.m} = \text{e} \rrbracket \} \\
\hline
\gamma \vdash \text{var.m} = \text{e} : \text{var.l'.m} = \text{e} \parallel \gamma' \parallel \text{C} \cup \text{C}' \quad (\text{assignAdd}) \\
\\
\frac{\gamma \vdash \text{e} : \text{e} \parallel \gamma' \parallel \text{C}' \quad \text{C} = \{ \llbracket \text{this.f} \rrbracket^\circ, \llbracket \text{e} \rrbracket \leq \llbracket \text{x.f} \rrbracket, \llbracket \text{ret.f} \rrbracket \leq \llbracket \text{new f(e)} \rrbracket \}}{\gamma \vdash \text{new f(e)} : \text{new f(e)} \parallel \gamma' \parallel \text{C} \cup \text{C}' \quad (\text{new})} \quad \frac{\gamma \vdash \text{e} : \text{e} \parallel \gamma' \parallel \text{C}' \quad \text{C} = \{ \llbracket \text{this.f} \rrbracket^\circ, \llbracket \text{e} \rrbracket \leq \llbracket \text{x.f} \rrbracket, \llbracket \text{ret.f} \rrbracket \leq \llbracket \text{f(e)} \rrbracket \}}{\gamma \vdash \text{f(e)} : \text{f(e)} \parallel \gamma' \parallel \text{C} \cup \text{C}' \quad (\text{funcCall})} \\
\\
\frac{\gamma \vdash \text{e} : \text{e} \parallel \gamma' \parallel \text{C}'}{\gamma \vdash \text{e.m} : \text{e.m} \parallel \gamma' \parallel \text{C} \cup \{ \llbracket \text{e} \rrbracket \leq [m : (\llbracket \text{e.m} \rrbracket, \bullet)] \} \quad (\text{memAcc})} \\
\\
\frac{\gamma \vdash \text{e}_1 : \text{e}_1 \parallel \gamma' \parallel \text{C}' \quad \gamma' \vdash \text{e}_2 : \text{e}_2 \parallel \gamma'' \parallel \text{C}'' \quad \text{C} = \{ \llbracket \text{e}_1 \rrbracket \leq [m : (\llbracket \text{e}_1.m \rrbracket, \bullet)], \llbracket \text{e}_2 \rrbracket \leq \llbracket \text{e}_1.m \rrbracket, \llbracket \text{e}_2 \rrbracket \leq \llbracket \text{e}_1.m = \text{e}_2 \rrbracket \}}{\gamma \vdash \text{e}_1.m = \text{e}_2 : \text{e}_1.m = \text{e}_2 \parallel \gamma'' \parallel \text{C} \cup \text{C}' \cup \text{C}'' \quad (\text{assignUpd})} \\
\\
\frac{\gamma \vdash \text{e}_1 : \text{e}_1 \parallel \gamma' \parallel \text{C}' \quad \gamma' \vdash \text{e}_2 : \text{e}_2 \parallel \gamma'' \parallel \text{C}'' \quad 1 \notin \gamma''(\text{lab}) \quad \text{C} = \{ \llbracket \text{e}_1 \rrbracket \leq [m : (\llbracket \text{e}_1.m \rrbracket, \bullet)], \llbracket \text{e}_1.m \rrbracket \leq ((\llbracket \text{call.this.l} \rrbracket \times \llbracket \text{call.x.l} \rrbracket) \rightarrow \llbracket \text{call.ret.l} \rrbracket), \llbracket \text{e}_1 \rrbracket \leq \llbracket \text{call.this.l} \rrbracket, \llbracket \text{e}_2 \rrbracket \leq \llbracket \text{call.x.l} \rrbracket, \llbracket \text{call.ret.l} \rrbracket \leq \llbracket \text{e}_1.m(\text{e}_2) \rrbracket \}}{\gamma \vdash \text{e}_1.m(\text{e}_2) : \text{e}_1.m(\text{e}_2) \parallel \gamma''[\text{lab} \mapsto (\gamma''(\text{lab}) \cup \{1\})] \parallel \text{C} \cup \text{C}' \cup \text{C}'' \quad (\text{methCall})} \\
\\
\frac{\gamma \vdash \text{e}_1 : \text{e}_1 \parallel \gamma' \parallel \text{C}' \quad \gamma' \vdash \text{e}_2 : \text{e}_2 \parallel \gamma'' \parallel \text{C}'' \quad \text{C} = \{ \llbracket \text{e}_2 \rrbracket \leq \llbracket \text{e}_1; \text{e}_2 \rrbracket \}}{\gamma \vdash \text{e}_1; \text{e}_2 : \text{e}_1; \text{e}_2 \parallel \gamma'' \parallel \text{C} \cup \text{C}' \cup \text{C}'' \quad (\text{seq})} \quad \frac{\gamma \vdash \text{e} : \text{e} \parallel \gamma' \parallel \text{C}' \quad \gamma'(x) = 1 \quad \text{C} = \{ \llbracket \text{e} \rrbracket \leq \llbracket \text{x.l} \rrbracket, \llbracket \text{e} \rrbracket \leq \llbracket \text{x.l} = \text{e} \rrbracket \}}{\gamma \vdash \text{x} = \text{e} : \text{x.l} = \text{e} \parallel \gamma' \parallel \text{C} \cup \text{C}' \quad (\text{varAss})}
\end{array}$$

Constraint Generation for Programs

$$\begin{array}{c}
\text{P} = \text{function } f_1(x) \{ e_1 \} \cdots \text{function } f_n(x) \{ e_n \} \\
\{ \text{this} \mapsto f_i, x \mapsto f_i, \text{lab} \mapsto \gamma'_{i-1}(\text{lab}) \} \vdash e_i : e_i \parallel \gamma'_i \parallel \text{C}_i \quad 1 \leq i \leq n \wedge \gamma_0 = \emptyset \\
\text{C} = \bigcup_{i \in 1..n} \text{C}_i \cup \{ \llbracket \text{e}_i \rrbracket \leq \llbracket \text{ret.f}_i \rrbracket \} \\
\hline
\vdash \text{P} : \text{C} \quad (\text{Prog})
\end{array}$$

Fig. 11. Constraint Generation

6.1 Constraint Closure

To simplify the extraction of a solution from a set of constraints we apply constraint closure, which makes the solutions (or lack of) explicit. The closing relation, $C \longrightarrow C'$, is defined in Figure 12⁸.

In (*closeTrans*) we add a constraint implied by the transitivity of subtyping.

In (*closeTransMem*) the type variable τ is required to have the same members as τ' with the same types definite annotation for m (because of $\tau \triangleleft_m \tau'$); the type variable τ' is required to have member m with type τ'' and annotation ψ (because of $\tau' \leq [m : (\tau'', \psi)]$). Therefore, τ is also required to have the member m with type τ'' and annotation ψ , as expressed by $\tau \leq [m : (\tau'', \psi)]$.

In (*closeBalance*) the type variable τ is required to be a subtype of τ' , and τ is required to be a subtype of a σ , *i.e.* either of `Int`, or of a function type. Because the subtype relationship for `Int` and function types is the identity, it follows that τ and σ will have to be “the same”, and therefore, it follows that τ' will have to be a subtype of σ .

In (*closeBalanceMem*) the type variable τ is required to have member m with type τ'' and annotation ψ . Because τ' is required to have the same members as τ with the same types it follows that τ' will also have the member m with type τ'' . The annotation, ψ , depends on whether $m = m'$. If $m = m'$ then ψ can be less defined *i.e.* \circ otherwise the annotations for m in τ and τ' must be the same.

In (*closeCong*) the same type variable, τ , is required to contain a member m with type τ' and also with type τ'' . It follows that τ' should be “equivalent” with τ'' . Similarly, in (*closeCongFunc*) because τ is required to be a subtype of two function types, it follows that the two function types should be “equivalent”, which, because of the subtype rules for function types, implies that the receiver, argument and return types should be “equivalent”.

Thus, for $[[\text{this.2}]] \triangleleft_{\text{add}} [[\text{this.1}]]$ and $[[\text{this.1}]] \leq [m\text{Sec} : ([[\text{this.1.mSec}], \bullet])]$ application of (*closeTransMem*) generates $[[\text{this.2}]] \leq [m\text{Sec} : ([[\text{this.1.mSec}], \bullet])]$, which ensures that $[[\text{this.2}]]$ will have member `mSec`. Also, closing $[[\text{this.2}]] \triangleleft_{\text{add}} [[\text{this.1}]]$ and $[[\text{this.2}]] \leq [\text{add} : ([[\text{this.2.add}], \bullet])]$ with (*closeBalanceMem*) generates $[[\text{this.1}]] \leq [\text{add} : ([[\text{this.2.add}], \circ])]$. Lastly, $[[\text{this.2}]] \leq [\text{add} : ([[\text{this.1.add}], \circ])]$ and $[[\text{this.2}]] \leq [\text{add} : ([[\text{this.2.add}], \bullet])]$, closed with rule (*closeCong*) generate $[[\text{this.1.add}]] \leq [[\text{this.2.add}]]$, and $[[\text{this.2.add}]] \leq [[\text{this.1.add}]]$.

Definition 4. C is closed, $\vdash C \diamond_{cl}$, if for any C' : $C \longrightarrow C'$ implies that $C = C'$.

Lemma 2 states that a set of constraints and its closure have the same set of solutions.

Lemma 2. If $S \vdash C$ and $C \longrightarrow C'$ then $S \vdash C'$.

⁸ We assume the closure of a set of constraints includes the reflexive closure.

$$\begin{array}{c}
\frac{c_1, \dots, c_n \longrightarrow c'_1, \dots, c'_m}{c_1, \dots, c_n \in \mathbf{C}} \text{ (closeMany1)} \quad \frac{}{\mathbf{C} \longrightarrow \mathbf{C}} \text{ (closeMany2)} \\
\\
\frac{}{\tau \leq \tau', \tau' \leq \rho \longrightarrow \tau \leq \rho} \text{ (closeTrans)} \\
\\
\frac{}{\tau \triangleleft \tau', \tau' \leq [\mathbf{m} : (\tau'', \psi)] \longrightarrow \tau \leq [\mathbf{m} : (\tau'', \psi)]} \text{ (closeTransMem)} \\
\\
\frac{}{\tau \leq \tau', \tau \leq \sigma \longrightarrow \tau' \leq \sigma} \text{ (closeBalance)} \\
\\
\frac{\psi' = \circ \text{ (if } \mathbf{m} = \mathbf{m}') \quad \psi' = \psi \text{ (otherwise)}}{\tau \triangleleft_{\mathbf{m}'} \tau', \tau \leq [\mathbf{m} : (\tau'', \psi)] \longrightarrow \tau' \leq [\mathbf{m} : (\tau'', \psi')]} \text{ (closeBalanceMem)} \\
\\
\frac{}{\tau \leq [\mathbf{m} : (\tau', -)], \tau \leq [\mathbf{m} : (\tau'', -)] \longrightarrow \tau' \leq \tau'', \tau'' \leq \tau'} \text{ (closeCong)} \\
\\
\frac{}{\tau \leq (\tau_1 \times \tau_2 \rightarrow \tau_3), \tau \leq (\tau'_1 \times \tau'_2 \rightarrow \tau'_3) \longrightarrow} \text{ (closeCongFunc)} \\
\tau'_1 \leq \tau_1, \tau_1 \leq \tau'_1, \tau'_2 \leq \tau_2, \tau_2 \leq \tau'_2, \tau_3 \leq \tau'_3, \tau'_3 \leq \tau_3
\end{array}$$

Fig. 12. Constraint Closure

6.2 Well-formed Constraints

The well-formedness of constraints, $\vdash \mathbf{C} \diamond$ (shown in Figure 13), ensures that a set of constraints can be used to create a solution. For a set of constraints, \mathbf{C} , to be well-formed they must be closed and all the constraints in \mathbf{C} must be well-formed. We define function $\mathcal{A}(\mathbf{C}, \tau, \mathbf{m})$ which determines the annotations that any solution satisfying \mathbf{C} should give to \mathbf{m} in τ . This is done by looking for constraints detailing members: $\tau \leq [\mathbf{m} : (-, -)]$. A member is annotated with \circ if there are no constraints indicating it should be definite:

$$\mathcal{A}(\mathbf{C}, \tau, \mathbf{m}) = \begin{cases} \bullet & \text{if } \tau \leq [\mathbf{m} : (-, \bullet)] \in \mathbf{C} \\ \circ & \text{if } \tau \leq [\mathbf{m} : (-, \circ)] \in \mathbf{C} \text{ and } \tau \leq [\mathbf{m} : (-, \bullet)] \notin \mathbf{C} \\ \mathcal{Udf} & \text{otherwise} \end{cases}$$

Intuitively, rule (*wlfNoDefs*) corresponds to the solution satisfaction rule (*solNoDefs*) in Figure 10. Where $\mathbf{S}(\tau)(\mathbf{m})$ is represented by looking for con-

straints detailing members, $\tau \leq [m : (-, -)]$, with $\mathcal{A}(\mathbf{C}, \tau, \mathbf{m})$ being used to find the appropriate annotation.

Rules (*wlfMix1*), (*wlfMix2*) and (*wlfMix3*) ensure that the constraints cannot mix object types with function types or integers.

$$\boxed{
\begin{array}{l}
\vdash \mathbf{C} \diamond_{cl} \\
\mathbf{C} = \{c_1 \dots c_n\} \\
\mathbf{C} \vdash c_i \quad \forall i \in 1 \dots n \quad (wlfAll) \\
\hline
\vdash \mathbf{C} \diamond
\end{array}
}$$

$$\frac{\tau \leq [m : (-, \bullet)] \notin \mathbf{C}}{\mathbf{C} \vdash \tau^\circ} (wlfNoDefs) \quad \frac{\tau \leq (- \times - \rightarrow -) \notin \mathbf{C} \wedge \tau \leq \text{Int} \notin \mathbf{C}}{\mathbf{C} \vdash \tau \leq [m : (\tau', \psi)]} (wlfMix1)$$

$$\frac{\tau \leq [m : (-, -)] \notin \mathbf{C} \wedge \tau \leq \text{Int} \notin \mathbf{C}}{\mathbf{C} \vdash \tau \leq (\tau_1 \times \tau_2 \rightarrow \tau_3)} (wlfMix2)$$

$$\frac{\tau \leq (- \times - \rightarrow -) \notin \mathbf{C} \wedge \tau \leq [m : (-, -)] \notin \mathbf{C}}{\mathbf{C} \vdash \tau \leq \text{Int}} (wlfMix3)$$

Fig. 13. Well-formed Constraints

6.3 From Constraints to Solutions

We now show how well-formed constraints, $\vdash \mathbf{C} \diamond$, can be translated into a solution. We first define a *type variable function*, \mathbf{V} , from type variables to variables in the type system, $\alpha_1 \dots \alpha_n \in \text{ObjVar}$. We say that \mathbf{V} is well-formed for \mathbf{C} , i.e. $\mathbf{C} \vdash \mathbf{V} \diamond$, iff $\tau \leq \tau', \tau' \leq \tau \in \mathbf{C}$ and $\mathbf{V}(\tau) = \alpha$ implies $\mathbf{V}(\tau') = \alpha$.

The translation relation, $\mathbf{C}, \mathbf{V}, \tau \rightarrow \text{tp}, \mathbf{V}'$, in Figure 14 translates a type variable, τ , into a type. If a type variable has no constraints indicating whether it should be an object, function or integer type, we default to making it an object type with no members. The extension of \mathbf{V} , which is denoted by $\mathbf{V} \oplus \tau$, is defined as follows (where α is a fresh variable):

$$(\mathbf{V} \oplus \tau)(\tau') = \begin{cases} \alpha & \text{if } \tau' \notin \text{dom}(\mathbf{V}) \text{ and} \\ & (\tau' = \tau \text{ or } (\tau' \leq \tau \in \mathbf{C} \text{ and } \tau \leq \tau' \in \mathbf{C})) \\ \mathbf{V}(\tau') & \text{if } \tau' \in \text{dom}(\mathbf{V}) \\ \text{Udf} & \text{otherwise} \end{cases}$$

$V \oplus \tau$ extends V with new type variables thus, keeping track of type variables that have already been translated. Because each step of the translation either extends V or finishes when $V(\tau) = \alpha$ (or $\text{tp} = \text{Int}$ or $\text{tp} = \mu \alpha.[\]$) termination is guaranteed.

		$\tau \leq [\mathbf{m} : -] \notin \mathbf{C}$ $\tau \leq (- \times - \rightarrow -) \notin \mathbf{C}$ $\tau \leq \text{Int} \notin \mathbf{C}$ $V' = V \oplus \tau$ $V'(\tau) = \alpha$
$\frac{V(\tau) = \alpha}{\mathbf{C}, V, \tau \rightarrow \alpha, V}$	$\frac{\tau \leq \text{Int} \in \mathbf{C}}{\mathbf{C}, V, \tau \rightarrow \text{Int}, V}$	$\frac{V'(\tau) = \alpha}{\mathbf{C}, V, \tau \rightarrow \mu \alpha.[\], V'}$
$n \geq 1$ $V(\tau) = \text{Udf}$ $V_0 = V \oplus \tau$ $V_0(\tau) = \alpha$ $\{\mathbf{m}_1 \dots \mathbf{m}_n\} = \{\mathbf{m} \mid \tau \leq [\mathbf{m} : (-, -)] \in \mathbf{C}\}$ $\tau \leq [\mathbf{m}_i : (\tau_i, -)] \in \mathbf{C} \text{ (for } i \in 1 \dots n)$ $\mathbf{C}, V_{i-1}, \tau_i \rightarrow \text{tp}_i, V_i$ $\psi_i = \mathcal{A}(\mathbf{C}, \tau, \mathbf{m}_i)$		$V(\tau) = \text{Udf}$ $V_0 = V \oplus \tau$ $V_0(\tau) = \alpha$ $\tau \leq (\tau_1 \times \tau_2 \rightarrow \tau_3) \in \mathbf{C}$ $\mathbf{C}, V_{i-1}, \tau_i \rightarrow \text{tp}_i, V_i \text{ (for } i \in 1 \dots 3)$
$\mathbf{C}, V, \tau \rightarrow \mu \alpha. [\mathbf{m}_1 : (\text{tp}_1, \psi_1) \dots \mathbf{m}_n : (\text{tp}_n, \psi_n)], V_n$		$\mathbf{C}, V, \tau \rightarrow \mu \alpha. (\text{tp}_1 \times \text{tp}_2 \rightarrow \text{tp}_3), V_3$

Fig. 14. Generating the Solution

6.4 Main Result

Lemma 3 states that if two type variables are “equivalent”, $\tau \leq \tau', \tau' \leq \tau \in \mathbf{C}$, by a set of well-formed constraints, they will translate to congruent types or the same variable.

Lemma 3. *If $\vdash \mathbf{C} \diamond$ and $\mathbf{C} \vdash V \diamond$ and $\mathbf{C}, V, \tau \rightarrow \text{tp}, V'$ and $\mathbf{C}, V, \tau' \rightarrow \text{tp}', V''$ and $\tau \leq \tau', \tau' \leq \tau \in \mathbf{C}$ then $\mathbf{C} \vdash V' \diamond$ and $\mathbf{C} \vdash V'' \diamond$ and $(\text{tp} \equiv \text{tp}' \text{ or } \exists \alpha : \text{tp} = \alpha = \text{tp}')$.*

Given a well-formed set of constraints and well-formed type variable function we define a generated solution, $S_{\mathbf{C}, V}$, such that $S_{\mathbf{C}, V}(\tau) = \mathbf{t}$ if and only if $\mathbf{C}, V, \tau \rightarrow \mathbf{t}, V'$. Theorem 4 states that a generated solution from a well-formed set of constraints is well-formed.

Theorem 4. *If $\vdash \mathbf{C} \diamond$ then $S_{\mathbf{C}, \emptyset} \vdash \mathbf{C}$.*

7 Related Work

Recursive Types and Subtyping Our choice of a recursive types was motivated by the need to allow typing of a large number of JavaScript programs, but at the same time make possible the development of a type inference algorithm. Hence, we have not considered more expressive type systems such as [19].

Type systems for object based languages have been developed mainly in a functional setting, see [1] and [14]. In [22] a type system is defined for the Abadi Cardelli object calculus with concatenation that uses recursive types. The definition of the object types is like ours (without function types) with width subtyping.

Subtyping for recursive function types (that are a subset of our types) has been considered in [4] where subtyping is contravariant on the input types and covariant on the return type. In our paper we have adopted congruence for subtyping between function types, because our aim is not to study the interaction between subtyping and recursive type (as in [4]) but to have a type system allowing type inference.

An imperative, type safe object oriented language, TOIL, was introduced in [8]. 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.

Dynamic Addition of Members Extensible objects are considered in a functional setting in [13]. An imperative calculus for extensible objects was proposed by Bono and Fisher, in [7]. In their 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 type system and their's is that we use recursive types (instead of row types plus universal and existential quantification). This makes it possible to have a decidable type inference algorithm. Note that, Bono and Fisher's aim was to encode classes in their object calculus, not to obtain a type inference algorithm.

In [24] Thiemann gives a type system for a considerable subset of JavaScript. Types are based on discriminative sums with two levels. The outer level determines what kind of base type e.g. number, string, object etc. The inner level determines the features of the type such as the value e.g. the singleton type `Number(100)`. Row types are used to detail the members of an object type. The type system models the automatic conversions that occur in JavaScript through a matching relation. As all conversions are tracked it is possible to flag those which could result in dangerous or unexpected behaviour. Access to a non-existent member does not result in a type error. There are no recursive types and no type inference algorithm is given but there is an implementation.

In the context of type assembly language Morrisett et al. in [17] uses an initialisation flag on the members of type to indicate if they have been assigned to. One could think of the potential and definite annotations of our types as representing the state of initialisation of a member.

Alias types are used in [5] and [9] to track the evolution of objects. In particular, in [9] 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.

Type Inference In [18,21] Palsberg et al. develop a type inference algorithm for a class based language based on flow analysis. The set of types is the class names defined in the program. Each expression, e , is given a type variable, $\llbracket e \rrbracket$, that expresses the - yet unknown - type. They employed a novel approach to model late binding through *conditional constraints*. A conditional constraint has the form $t \in \llbracket e \rrbracket \implies C$ saying that constraints C are only applicable when t is a possible type for $\llbracket e \rrbracket$. In [3] this work is applied to the object based language SELF[12]. Each occurrence of an object is given a unique token ω . Thus, object structure is derived from the program. Our work differs in that we must infer the structure of objects. With StarKiller[23] Salib uses the Cartesian Product Algorithm[2] to infer types for Python programs in order to improve compiled code. Object types maintain a reference to their definition when a member is added or updated new object types are generated and propagated through the system.

In [20] Palsberg considers type inference for the first order type system (with recursive types and subtyping) for the Abadi Cardelli object calculus[1]. The system of constraints is a subset of those used in this paper, with two kinds $\tau \leq \tau'$ and $\tau \leq [m : (\tau, \psi)]$. Furthermore, the Abadi Cardelli calculus does not allow member addition like JS₀. The type system uses a subsumption rule which is encoded in the system by having two type variables for each program point, one before subtyping and one after. For variables there is x and $\llbracket x \rrbracket$ and member access, $\llbracket e.m \rrbracket$ and $\langle e.m \rangle$. Instead of a subsumption rule, our type system uses the subtype relation where necessary *e.g.* the actual parameter being a subtype of the formal parameter. Hence, the subtype relation is always used *explicitly* between the types of expressions in the program. Therefore, we don't need to use two type variables to model the application of subsumption. After the constraints are generated a graph is generated and closed. A well-formedness criteria is given to graphs which are then converted to an automata which is used to annotate the program. Our work differs in that we specify closure and well-formedness in terms of constraints rather than convert to a graph.

In [10] Eifrig et al. consider type inference for the class-based language *I-LOOP*. The types are *recursively constrained* in that a type is supplemented with a set of constraints, $\tau \setminus C$. They take a different approach to us by defining type rules that generate constraints and then modifying the rules to make a deterministic and complete inference system. Fields and methods of a type are detailed with constraints of the form $\tau \leq \mathbf{Inst} \ m : \tau'$, which states that τ has a field m of type τ' .

In [25], Wang et al. give a type inference system for Java that can statically verify the correctness of downcast. The types used are based on those used in [10] as described above. There are types that describe the structure of ob-

jects, **obj** ($\delta, \overline{\mathbb{1}_i : \tau_i}$) where δ and $\mathbb{1}_i$ are abstract labels for the class name and fields/methods respectively. The structure of the object types is derived from the class structure. Unlike our treatment of method call sites, where we *always* allocate new type variables, they delegates this to closure. By parameterizing closure with a mapping it is possible to *share* type variables between different invocations of a method.

8 Conclusions and Further Work

In this paper a flexible type system for an idealized version of JavaScript is presented, its soundness is outlined, and a type inference algorithm for this type system is defined. The type inference algorithm is sound with respect to the type system. We show how well-formed constraints can be used to generate a solution and annotate an untyped JS₀ program. The main challenges for both the type system and the inference are the imperative nature of the language combined with the possibility of extending objects.

For future work we want to study the completeness of the type inference algorithm, its complexity, and extend the type system to allow more typeable expressions, e.g., allowing a more flexible subtyping for functions. To show completeness we need principality of the type produced, this is quite difficult to achieve for recursive type systems. We would also like to develop a *mixed mode* system where some of the type annotations are already given by the user. For example, we could provide a typing of the Document Object Model[11] and check code in web pages against it.

9 Acknowledgements

We would like to thank Mario Coppo, Mariangiola Dezani, Matthew Smith and Alex Buckley for their help and insight and our colleagues at Imperial College Department of Computing and Dipartimento di Informatica of Torino University. We would also like to thank the anonymous ECOOP reviewers.

References

1. Martín Abadi and Luca Cardelli. *A Theory of Objects*. Springer-Verlag, New York, NY, 1996.
2. Ole Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference of Parametric Polymorphism. In *ECOOP*, 2-26, 1995.
3. Ole Agesen, Jens Palsberg, and Michael I. Schwartzbach. Type inference of SELF: Analysis of objects with dynamic and multiple inheritance. *Softw., Pract. Exper.*, 25(9):975–995, 1995.
4. Roberto M. Amadio and Luca Cardelli. Subtyping recursive types. *ACM Transactions on Programming Languages and Systems*, 15(4):575–631, 1993.
5. 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.

6. Christopher Anderson and Paola Giannini. Type checking for javascript. In *WOOD '04*, volume WOOD of *ENTCS*. Elsevier, 2004. <http://www.binarylord.com/work/js0wood.pdf>.
7. 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.
8. 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.
9. F. Damiani and P. Giannini. Alias types for environment aware computations. In *WOOD '03*, volume 82 of *ENTCS*. Elsevier, 2003.
10. Jonathan Eifrig, Scott F. Smith, and Valery Trifonov. Sound polymorphic type inference for objects. In *Proc. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)'95*, pages 169–184, New York, NY, 1995. ACM Press.
11. Arnaud Le Hors et al. Document Object Model (DOM) Level 3 Core Specification. Technical report, 1998. <http://www.w3.org/TR/2003/CR-DOM-Level-3-Core-20031107>.
12. Ole Agesen et al. The SELF 4.0 Programmer's Reference Manual. <http://research.sun.com/self/>, 1995.
13. 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.
14. 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*.
15. David Flanagan. *JavaScript - The Definitive Guide*. O'Reilly, 1998.
16. ECMAScript Language Specification. ECMA International. ECMA-262, 3rd edition, december 1999. <http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf>.
17. Greg Morrisett, David Walker, Karl Crary, and Neal Glew. From system f to typed assembly language. *ACM Trans. Program. Lang. Syst.*, 21(3):527–568, 1999.
18. Nicholas Oxhoj, Jens Palsberg, and Michael I. Schwartzbach. Making type inference practical. In *ECOOP*, pages 329–349, 1992.
19. W. Hill W. Olthoff P. Canning, W. Cook and J. C. Mitchell. F-bounded polymorphism for object-oriented programming. In *Proc. Conf. on Functional Programming Languages and Computer Architecture*, pages 273–280. ACM Press, 1989.
20. Jens Palsberg. Efficient inference of object types. *Inf. Comput.*, 123(2):198–209, 1995.
21. Jens Palsberg and Michael I. Schwartzbach. Object-oriented type inference. In Norman Meyrowitz, editor, *Proceedings of the Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA)*, volume 26, New York, NY, 1991. ACM Press.
22. Jens Palsberg and Tian Zhao. Type inference for record concatenation and subtyping. *Inf. Comput.*, 189(1):54–86, 2004.
23. Mike Salib. Static Type Inference (for Python) with Starkiller. <http://www.python.org/pycon/dc2004/papers/1/paper.pdf>, 2004.
24. Peter Thiemann. Towards a type system for analyzing javascript programs. In *ESOP*, pages 408–422, 2005.
25. Tiejun Wang and Scott F. Smith. Precise constraint-based type inference for java. In *ECOOP '01: Proceedings of the 15th European Conference on Object-Oriented Programming*, pages 99–117. Springer-Verlag, 2001.