Polymorphic Type Inference for Scripting Languages with Object Extensions

Tian Zhao

University of Wisconsin – Milwaukee,
Milwaukee, Wisconsin, USA
tzhao@uwm.edu

Abstract

This paper presents a polymorphic type inference algorithm for a small subset of JavaScript. The goal is to prevent accessing undefined members of objects. We define a type system that allows explicit extension of objects through add operation and implicit extension through method calls. The type system also permits strong updates and unrestricted extensions to new objects. The type inference algorithm is modular so that each function definition is only analyzed once and larger programs can be checked incrementally.

Categories and Subject Descriptors F.3.2 [*Logics and Meanings of Programs*]: Semantics of Programming Languages – program analysis; F.3.3 [*Logics and Meanings of Programs*]: Studies of Program Constructs – type structure

General Terms Languages, Theory, Verification

Keywords Type inference, static types, JavaScript, dynamic languages

1. Introduction

JavaScript is an object-based scripting languages widely used for Web applications. As a dynamic language, JavaScript allows objects be modified at runtime for updating existing members, adding new members, or deleting members. Functions can be added to objects to become methods. Object members can be replaced by values of different types. This dynamic nature can result in runtime errors such as accessing an undefined object member or invoking an object property that is not a function. Since JavaScript is dynamically typed, there is no static approach to automatically determine the type information of objects at each program point and runtime type tests are often used to examine the properties of objects.

In JavaScript, objects may be created directly as object literals or by invoking constructor functions through *new* operator. After an object is created, it can be extended with new members. The extension may be explicit through assignment. For example the statements below extend empty object `x` with a new member of number type.

```
var x = {};
x.m = 1;
```


If we write the initial type of `x` as $[\,]$, then its type at line 2 becomes $[\texttt{m} : \texttt{num}]$, where $[\,]$ represents the type of empty object and `num` is the type of numbers. The extension may also be implicit through function calls.

```
var x = {};
function f(y) {
    y.m = 1;
}
f(x);
var z = x.m;
```

After the call to `f`, the type of `x` also has member `m`.

Because of this kind of extensions, an object may have different sets of members during execution and a variable pointing to the object should be assigned object types to reflect the correct set of members at each program point. A type inference algorithm has to statically track the members added to the object variable at different parts of a program.

To address this problem, type inference algorithms [4, 12, 25] were developed to statically track the members of objects. However, these type systems are rather restrictive in that they infer monomorphic types and whole program analysis is needed for each program. These type systems may also have difficulty in dealing with cases such as polymorphic functions and container objects. A more useful type inference algorithm needs to support both object extensions and polymorphism.

Polymorphism promotes code reuse and is an essential part of object-oriented programming. JavaScript functions exhibit parametric polymorphism when invoked with different types of arguments. The arguments to a JavaScript function include an implicit self pointer accessible via *this* variable. A function may be assigned as methods to different types of objects and if the function is invoked as a method, then the self pointer of the function is bound to the receiver object of the call. It is too restrictive to assign monomorphic type to a function since different arguments to the function are forced to be have common super types.

For example, the constructor function `F` shown below can be used to create different types of objects.

```
function F(x) {
  this.left = x;
}
var a = new F(1);
var b = new F(true);
```

The type for `F` can be $t \rightarrow [\texttt{left} : t]$ so that the types of `a` and `b` are $[\texttt{left} : \texttt{num}]$ and $[\texttt{left} : \texttt{bool}]$ respectively.

However, subtyping can complicate type inference. The function `copy` shown below requires an object with member `left` and return the object after adding or updating a member `right`.

```
function copy(x) {
   x.right = x.left;
   return x;
}
var a = new F(1);
var b = new F(1);
b.middle = 0;

var a1 = copy(a);
var b1 = copy(b);
```

It is not immediately clear how to assign a type to `copy` to support polymorphism. By Anderson et al.'s design [4], we can assign a type $\tau \to \tau'$ to the function `copy`, where

$$\tau = [\texttt{left} : (\mathsf{num}, \bullet), \texttt{right} : (\mathsf{num}, \circ)]$$
$$\tau' = [\texttt{left} : (\mathsf{num}, \bullet), \texttt{right} : (\mathsf{num}, \bullet)]$$

and $\bullet$ labels definitely present members while $\circ$ labels potential members of objects. The types of `a` and `b` must have potential member `right` as well since they are subtypes of the parameter type of `copy`. There are two problems with this approach. One is that the return type of the constructor `F` is forced to have a potential member `right` of `num` type, which is unnecessarily restrictive. Another problem is that the statement

```
var c = b1.middle;
```

would not be typable even though it is safe. Recency type system [12] may also have similar problem with this example as it needs to assign a summary object type (that have a fixed set of members) to the parameter of `copy` since it is not a precise object.

To address the above problems, we define a polymorphic, constraint-based type inference algorithm for a small subset of JavaScript with support for dynamic extension and update of object members.

We can avoid the first problem by assigning a kind of extensive object types called *pro-type* to new objects (e.g. `a` and `b` in previous example) so that their types can be extended with new members when necessary but not before that (e.g. by function call `copy(a)` and `copy(b)`). This design is similar to the singleton object type in recency type system though pro-types are only for local variables.

Specifically, we use two kinds of object types: pro-type and obj-type. The pro-types are assigned to local variables that refer to objects created in the local scope and we allow strong updates (where members of an object can be replaced by values of different types) and unrestricted extensions to the variables of pro-types. The obj-types are assigned to other local variables, function parameters, and object fields. A pro-type is uniquely associated with an object while an obj-type may be related to multiple objects. A variable of obj-type can also be extended though it may not have strong updates. Our type system also keeps track of members added to an object by implicit extension through function parameters and self pointer. The latter is called self-inflicted extension [7] – the extension that an object made to itself upon receiving a message.

For the second problem, our algorithm uses a generalized form of types adopted from the recursively constrained (rc) types [8], which was used for polymorphic type inference for an object-oriented language. The rc types have the form of $\tau \setminus C$, where C is a set of type constraints, each of the form $\tau_1 \leq \tau_2$. The rc type is a generalization of recursive types and it can model a restricted form of union and intersection type with multiple upper bounds and lower bounds on the same type variable. The type that we infer for the function `copy` is parameterized so that each application of `copy` results in a new version of the type. As a result, the call `copy(b)` returns an object with a member `middle` while `copy(a)` does not.

Our system has its limitations. We do not consider JavaScript features such as accessing objects as associative arrays, function prototypes, variadic functions, and eval function. Also, we do not allow strong update of object types through function or method calls. For example, the last line in the code fragment below is not typable even though it will run correctly.

```
var d = new F(1);
d.right = true;
var d1 = copy(d);
var e = d1.right + 1;
```

That is, we cannot overwrite the boolean type of the `right` member of `d` when it is assigned the integer value of its `left` member in function `copy`. In fact, the type of the `d1.right` is inferred as the super type of both `num` and `bool` so that `d1.right + 1` is not typable. We will consider these limitations in future work.

In summary, we present a type inference algorithm for a small subset of JavaScript language that

1. infers polymorphic function types,
2. keeps track of object extensions through direct assignments and function/method calls, and
3. allows strong updates and flexible extensions to new objects in local scope.

In the rest of paper, we first present some motivating examples in Section 2 and then we give an informal discussion of our approach in Section 3. Next, we formalize a type system on a language that models some core features of JavaScript. We present the syntax and typing rules in Section 4. We explain the type inference algorithm and the simplification of the inferred types in Section 5. The operational semantics and soundness proof of the type system are in the appendix.

2. Motivating examples

```
1 function Point(x, y) {
2   this.x = x;   this.y = y;
3 }
4 function Line(p1, p2) {
5   this.p1 = p1;   this.p2 = p2;
6 }
7 function leftMost(z1, z2) {
8   var x1 = z1.getX(), x2 = z2.getX();
9   var r;
10  if (x1 <= x2)  r = z1; else r = z2;
11  return r;
12 }
13 function getX() {  return this.x; }
14
15 function getLineX() {
16   var p1 = this.p1; p1.getX = getX;
17   var p2 = this.p2; p2.getX = getX;
18
19   /* return a point */
20   var p = leftMost(p1, p2);
21   return p.x;
22 }
23 p1 = new Point(1, 2);
24 p2 = new Point(3, 1);
25 line  = new Line(p1, p2);
26
27 p2.getX = getX;
28 line.getX = getLineX;
29
30 /* return a point or line */
31 s = leftMost(line, p2);
```

Figure 1. Object extension and polymorphism

Figure 1 is an example of object extension and polymorphism, where `Point` and `Line` are constructor functions that return new point and line objects respectively. The function `leftMost` takes two geometry objects (either line or point) and return the one on the left based on the left-most x-coordinate of the geometry. The function assumes that the arguments have a `getX` method that returns the left-most x-coordinate. Before calling the `leftMost` function with a point or line object as argument, it is necessary to add appropriate `getX` methods to the geometries. For example, before the call `leftMost(line, p2)` on line 31, function `getX` is added to the point object `p2` (line 27) and `getLineX` is added to the `line` object (line 28).

Notice that the call `leftMost(line, p2)` returns either a point or a line object. However, in function `getLineX` , the function call `leftMost(p1, p2)` (line 20) returns a point object. The statement at line 21 assumes the returned object is a point and read its member `x`. This polymorphic use of function `leftMost` makes it difficult to assign static types to the parameter and return types. For example, we cannot assign a Point type or Line type to the parameters of `leftMost` since it accepts both kinds of argument. It is possible that the function may take other kinds of argument such as polygon. Parametric function type does not work in this case either. If we assign variable types α_1 and α_2 to the function parameter `z1` and `z2` respectively, then the return type is neither α_1 nor α_2 but an upper bound of the union type $\alpha_1 \vee \alpha_2$.

```
1 function average(p1, p2) {
2    var x = (p1.x + p2.x)/2;
3    var y = (p1.y + p2.y)/2;
4
5    return new Point(x,y);
6 }
7 function Line(p1, p2) {
8   this.p1 = p1;   p1.geom = this;
9   this.p2 = p2;   p2.geom = this;
10
11  this.center = average(p1, p2);
12 }
13
14 p1 = new Point(1, 2); p2 = new Point(3, 1);
15
16 /* implicit extension to p1 and p2 */
17 line  = new Line(p1, p2);
18
19 /* p1 now has geom field */
20 g = p1.geom;
21
22 function center() {
23    var p1 = this.p1;
24    var p2 = this.p2;
25    return average(p1, p2);
26 }
27 line.center = center;    /* strong update */
28
29 p = line.center();
```

Figure 2. Implicit extension and strong update

JavaScript objects may be extended after being passed as arguments to functions or as receivers of method calls. Figure 2 shows such an example, where point objects passed to the `Line` constructor are extended with a field `geom` that points to the line object (line 8 and 9). After call to the `Line` constructor, the point `p1` has a field `geom` and its access is allowed (line 20).

Figure 2 also includes an example of strong update. The `Line` constructor adds a field `center` to each new line object, which is the center point of the line. If a line's end points may change, it may be better to have a method to compute the center point based on the current end points. To do this, we simply replace the `center` member of the `line` object with a function `center` at line 27. This is strong update since the new member is a function while the existing member is an object.

3. Approach

In this section, we give an informal description of our type system using the motivating examples.

3.1 Function types

In our design, we assume that a program consists of a sequence of function declarations (including constructor functions) and a main program (consists of a sequence of statements). For each function of the form

$$\mathsf{function}\ f(x)\{\langle\text{function body}\rangle\},$$

we infer a rc type

$$\tau \setminus C,$$

where τ is the type of the function and C is a set of constraints that constrain τ and other types related to τ.

The set C is formed through the type derivation of the statements of the function body. After each statement is analyzed, C may be added with more constraints. The set C must be consistent after each statement is analyzed so that its closure does not contain inconsistent constraints such as $\mathsf{num} \leq \mathsf{bool}$. We do not directly verify subtyping relation between types. Instead, to check whether $\tau_1 \leq \tau_2$, we add this constraint to the current constraint set C and check if $C \cup \{\tau_1 \leq \tau_2\}$ is consistent. One problem with this approach is that a large number of constraints may be generated for a function while many of them are not needed after the function is checked. To reduce the size of constraint set, we define simplification rules to remove redundant constraints from C and only keep those constraints related to the parameter types and return types of the functions.

For the rest of the section, we will refer to C as the constraint set of the current context.

To support parametric polymorphism, for each reference of a function/constructor of the type $\tau_f \setminus C_f$, we instantiate the type by replacing the free variables in τ_f and C_f with fresh variables using a renaming function Ψ and add ΨC_f to C.

3.2 Object types

We have two kinds of object types: pro-type and obj-type. A pro-type (denoted by meta variable ς) is uniquely associated with an object such as the *self* object of a constructor function or a newly instantiated object.

All pro-types are initially type variables. We use meta variable β for the initial type of the *self* object of a constructor function and use meta variable γ for the initial type of a new object. If a variable y of type ς is explicitly extended through assignment such as $y.m = y'$, we give y a new pro-type

$$\varsigma \leftarrow (m, t),$$

where the constraint $\tau \leq t$ is added to C and τ is the type of y'. The type $\varsigma \leftarrow (m, t)$ has the meaning that ς is added a member m of type t and if this member already exists in ς, then it is replaced. This representation permits strong update.

In the `Point` function in Figure 1, the initial type of `this` variable is a variable pro-type β_p that represents empty object. After line 2, the type of `this` becomes

$$\varsigma_p = (\beta_p \leftarrow (x, t_1)) \leftarrow (y, t_2),$$

with the constraint set $\{t_x \leq t_1, t_y \leq t_2\}$, where t_x, t_y are parameter types of the constructor. The rc type of `Point` is then

$$t_x \times t_y \rightarrow \varsigma_p \setminus \{t_x \leq t_1, t_y \leq t_2\}.$$

We let the type constructor $\leftarrow$ be left associative so that we can write $(\beta_p \leftarrow (x, t_1)) \leftarrow (y, t_2)$ as $\beta_p \leftarrow (x, t_1) \leftarrow (y, t_2)$.

If we instantiate a point object as below

```
var p = new Point(1,2);
```

then the type of `p` is γ_p with the constraints

$$\{\beta'_p \leftarrow (\mathtt{x}, t'_1) \leftarrow (\mathtt{y}, t'_2) \leq \gamma_p, \\ \mathsf{num} \leq t'_x, t'_x \leq t'_1, \mathsf{num} \leq t'_y, t'_y \leq t'_2\}.$$

The type γ_p is a pro-type variable representing the initial type of the new object `p` and the primed type variables are instances of the free variables in the rc type of `Point`.

An obj-type may be related to multiple objects and is also initially a type variable (denoted by meta variable t). We give obj-types to function parameters, object members, and the local variables that are not uniquely associated with an object. An object of obj-type may be extended and we write the type of the extended object as (t, M), where M is a meta variable representing the names of members added to t. If a variable y of an obj-type t is explicitly extended in $y.m = y'$, then the type of y becomes $(t, \{m\})$, which means that m is a member added to t but the type of m is not changed. Also, the constraints

$$\{t \leq [m^- : t'], \tau \leq t'\}$$

are added to C, where τ is the type of y' and constraint of the form $t \leq [m^- : \_]$ is added for each write on obj-types.

3.3 Reading an object member

In general, we add constraints of the form

$$\{\tau \leq [m^+ : t], m \in \tau\},$$

for each read of a member m (including method call) on an object of the type τ, where $m \in \tau$ is a special type of constraints that check whether τ or its type lower bounds (if τ is a type variable) have the member m. A constraint of the form $m \in \beta$ is immediately inconsistent since β represents an empty object.

For example, the rc type of the function `getX` in Figure 1 is

$$t \rightarrow t_g \setminus \{t \leq [\mathtt{x}^+ : t_g], \mathtt{x} \in t\},$$

where t is the type of `this` variable in `getX`. The constraints $\{t \leq [\mathtt{x}^+ : t_g], \mathtt{x} \in t\}$ correspond to reading the member `x` on `this` variable. Since t is a type variable, its lower bounds must have the member `x`. Within the rc type of `getX`, t has no lower bound but when `getX` is invoked on an object, the lower bounds of t must have a member `x`.

Consider the code fragment

```
var p = new Point(1,2);
p.getX = getX;
var x = p.getX();
```

for `p.getX = getX`, our type system instantiates the type of `getX` to $t' \rightarrow t'_g$ and add $\{t' \leq [\mathtt{x}^+ : t'_g], \mathtt{x} \in t'\}$ to C. For `p.getX()`, the type system adds some constraints to C so that its closure includes a constraint $\gamma_p \leq t'$, where γ_p is the type of `p` as defined previously. By transitivity, $\beta' \leftarrow (\mathtt{x}, t'_1) \leftarrow (\mathtt{y}, t'_2)$ is a lower bound of t' and it has the member `x`.

3.4 Polymorphism

To illustrate parametric polymorphism, consider the `leftMost` function invoked on line 20 and line 31 of Figure 1. The type of the function can be written as

$$\kappa = t_1 \times t_2 \rightarrow t \setminus \begin{array}{l} \{t_1 \leq [\mathtt{getX}^+ : t_3], t_2 \leq [\mathtt{getX}^+ : t_4], \\ \mathtt{getX} \in t_1, \mathtt{getX} \in t_2, \\ t_3 \leq t_5 \rightarrow t_7, t_4 \leq t_6 \rightarrow t_8, \\ t_1 \leq t_5, t_2 \leq t_6, t_1 \leq t, t_2 \leq t \\ t_7 \leq \mathsf{num}, t_8 \leq \mathsf{num}\}. \end{array}$$

Each time `leftMost` is called, we instantiate κ by renaming its free variables t and t_1 – t_8 to some fresh variables.

To assign types for the statement `var p = leftMost(p1, p2)` at line 20 (within the function `getLineX`), we instantiate κ to

$$t'_1 \times t'_2 \rightarrow t' \setminus C', \text{ where } \{t'_1 \leq t', t'_2 \leq t'\} \subseteq C'.$$

Assume the types of `p1` and `p2` are τ_1 and τ_2 respectively. Then

$$C' \cup \{\tau_1 \leq t'_1, \tau_2 \leq t'_2\} \subseteq C.$$

At line 21, the statement `p.x` adds a constraint $\mathtt{x} \in t'$ and since τ_1 and τ_2 are lower bounds of t', they need to have the member `x`. This is satisfied since `getLineX` is used as a method of a line object so that `p1` and `p2` are end points of the line.

To assign types for the statement `s = leftMost(line, p2)` at line 31, we instantiate κ to

$$t''_1 \times t''_2 \rightarrow t'' \setminus C'', \text{ where } \{t''_1 \leq t'', t''_2 \leq t''\} \subseteq C''.$$

The types of `line` and `p2` are the lower bounds of t'', which is the type of the variable `s`. Consequently, we cannot treat `s` as a point or line. We can only access members in `s` that are defined in both objects such as `getX`.

3.5 Implicit extension

Objects may be extended implicitly when they are passed as arguments to function calls. As an example, consider the constructor `Line` in Figure 2 that extends its two parameters with a field `geom` that points to the line object being constructed. After a line object is instantiated (line 17), the arguments `p1` and `p2` should have a new field `geom`. To model this behavior, we write the type of `Line` as

$$(t_1, \{\mathtt{geom}\}) \times (t_2, \{\mathtt{geom}\}) \rightarrow \varsigma \setminus \begin{array}{l} \{\ t_1 \leq [\mathtt{geom}^- : t_{\mathtt{g1}}], \\ \ \ t_2 \leq [\mathtt{geom}^- : t_{\mathtt{g2}}], \\ \ \ t_1 \leq t_{\mathtt{p1}}, t_2 \leq t_{\mathtt{p2}}, \\ \ \ \varsigma_1 \leq t_{\mathtt{g1}}, \varsigma_2 \leq t_{\mathtt{g2}}, \ldots\ \} \end{array}$$

where

$$\begin{array}{rcl} \varsigma_1 & = & \beta \leftarrow (\mathtt{p1}, t_{\mathtt{p1}}), \\ \varsigma_2 & = & \varsigma_1 \leftarrow (\mathtt{p2}, t_{\mathtt{p2}}), \\ \varsigma & = & \varsigma_2 \leftarrow (\mathtt{center}, t). \end{array}$$

For clarity, the constraints related to the addition of the member `center` on line 11 are omitted.

In general, a function type is written as

$$(t, M) \times (t', M') \rightarrow \tau \setminus C$$

where M and M' represent the set of members extended through the parameters. In this example, the extended member is `geom`. At line 17, if the type of `p1` is τ_1 before the call, then its type becomes $(\tau_1, \{\mathtt{geom}\})$ after the call and accessing this member on `p1` is allowed (line 20).

3.6 Strong update

A variable of pro-type can receive strong update to replace an existing member with one of different type. In Figure 2, the assignment `line.center = center` (line 27) replaces the existing `center` (a point object) of a line object with a function. If the line object at line 27 has a type ς_l, then after the assignment, the type becomes

$$\varsigma_l \leftarrow (\mathtt{center}, t), \text{ where } \tau_c \leq t \in C,$$

and τ_c is an instance of the type of the function `center`.

4. Formalization

In this section, we present a formalization of our type system. We first explain the syntax and the typing rules. The type inference algorithm is given in Section 5. The operational semantics and type soundness proof are in the appendix.

4.1 Syntax

We select a small subset of the JavaScript language that includes member select, member update/add, method calls, object creation, and branch statement with syntax shown in Figure 3. We distinguish constructor function and regular function with the naming convention that constructor function name starts with an upper case letter. We do not include function calls since its behavior is similar to that of method calls when the receiver object is empty. In fact, regular function calls in JavaScript will substitute `this` pointer of the called function with the global object [6].

P	::=	${Fn_i}^{i\in 1..n}\ s$	Program
Fn	::=	$\mathsf{function}\ f(x)\{s; \mathsf{return}\ y\}$	function
	|	$\mathsf{function}\ F(x)\{s\}$	constructor
s	::=		statements
		$x = z$	assignment
	|	$x = \mathsf{new}\ F(y')$	new object
	|	$x = y.m$	member select
	|	$x = y.m(y')$	method call
	|	$y.m = y'$	member update/add
	|	$s; s'$	sequence
	|	$\texttt{if}(x)\ \{s\}\ \texttt{else}\ \{s'\}$	if statement
y	::=	x	local variables
	|	`this`	self reference
z	::=	y	
	|	f	function identifier
	|	n	number
	|	b	boolean
	|	null	null value

Figure 3. Syntax

The syntax of a function body consists of a sequence of statements and a return statement. For simplicity, we write object creation. member select, and method call in the form of assignments and each expression is assigned to a variable so that there is no nested expressions in the statements. We omit the *var* declaration of variables with the understanding that variables appearing within a function are locally scoped. The body of a constructor function has a sequence of statements but no return statement since each time a constructor function is called through new operator, `this` pointer of the function is given a new empty object and after the body is executed, `this` object is returned.

The meta variable f ranges over the names of regular functions, F ranges over the names of constructor functions, and m ranges over member names. A program P consists of a one or more function/constructor definitions and a main statement s.

4.2 Types

τ	::=	ς	pro-type
	|	$\overline{(t, M)} \to \tau$	function type
	|	t	type variable (obj-type)
	|	(t, M)	extended type
	|	num | bool	base type
	|	null	null type

The meta variable τ ranges over type variables, function types, extended types, pro-types, primitive types (num and bool), and null type. A type variable t may be given to a variable that references any kind of values and when the value is an object, then t is considered an obj-type. An extended type (t, M) is assigned to an obj-type variable after it is extended. In a function type

$$\overline{(t, M)} \to \tau,$$

we write $\overline{(t, M)}$ to represent the types of one or more parameters and it is either $(t_1, M_1) \times (t_2, M_2)$ for a function (where t_1 is the type of the self pointer) or (t, M) for a constructor. In the previous examples, we omit the member set M in the parameter types whenever it is empty.

ς	::=	β	empty pro-type
	|	γ	initial instance pro-type
	|	(ς, M)	implicitly extended pro-type
	|	$\varsigma \leftarrow (m, t)$	explicitly extended pro-type
	|	(ς, ς')	merged pro-type

The pro-types are given to variables that reference new objects. The self pointer of a constructor is initially given variable type β that represents empty object. A pro-type ς maybe extended directly by assignment to $\varsigma \leftarrow (m, t)$ or indirectly through function/method calls to (ς, M). If a variable has different pro-types ς and ς' in the two branches of an *if* statement, then we merge the types into (ς, ς'). When a new object is instantiated, the variable that holds the new object is given a pro-type γ. The closure of the current constraint set will include a constraint of the form $\varsigma \leq \gamma$, where ς is the return type of the constructor. We distinguish this type from the empty pro-type β since a constraint of the form $\texttt{m} \in \beta$ is immediately inconsistent (which indicates that the member `m` is not defined) while $\texttt{m} \in \gamma$ is not. For example, given the statement `p1 = new Point(1, 2)`, we assign a type γ to `p1`, so that the current constraint set includes $\varsigma \leq \gamma$, where $\varsigma = \beta \leftarrow (\texttt{x}, t_1) \leftarrow (\texttt{y}, t_2)$ is an instance of the return type of `Point`.

The meta variable M represents a set of member names and it ranges over set variable $\mathcal{M}$, empty set, a singleton set $\{m\}$, the union or intersection of two sets. We use set variable $\mathcal{M}$ in the typing rule for method calls to model the implicit extension of method arguments.

M	::=	$\mathcal{M}$	member set variable
	|	$\emptyset$	
	|	$\{m\}$	
	|	$M \cup M'$	
	|	$M \cap M'$	

A recursively constrained type κ has the form of $\tau \setminus C$. where C consists of constraints of the form:

τ	$\leq$	$[m^{\psi} : t]$,	where $\psi \in \{+, -\}$
τ	$\leq$	τ',	where $\tau' \neq (t, M), \tau' \neq \mathsf{null}$
m	$\in$	τ	
$\mathcal{M}$	$\subseteq$	M	

In addition, if a function type τ' appears in $\tau \leq \tau'$, then τ' has the form $\overline{(t, \mathcal{M})} \to t$.

4.3 Constraint closure

The definition of a closed set of constraints is based on the closure rules in Figure 4. The rules (C1) to (C5) check type constraints while the rules (M1) to (M4) are for member constraints. We write $\mathsf{cls}(C)$ to represent the closure of a constraint set C.

Rule (C1) and (C2) propagate type upper bound w that includes base types, function types, and field types (of the form $[m^{\psi} : t]$

$$
\begin{array}{lrcl}
(C1) & \{\tau \le t, t \le w\} \subseteq C & \Rightarrow & \tau \le w \in C \\
(C2) & (t, M) \le w \in C & \Rightarrow & t \le w \in C \\
(C3) & \overline{(t, M)} \to \tau \le \overline{(t', \mathcal{M})} \to t \in C & \Rightarrow & \{\overline{t' \le t}, \overline{\mathcal{M} \subseteq M}, \tau \le t\} \subseteq C \\
(C4) & t \in \mathsf{field}_C(\varsigma, m), \{\tau \le t, \varsigma \le [m^+ : t']\} \subseteq C & \Rightarrow & \tau \le t' \in C \\
(C5) & \varsigma \le t \in C,\ \varsigma\ \mathsf{derive}_C\ \varsigma' & \Rightarrow & \varsigma' \le t \in C \\
(M1) & \{m \in \tau, \tau' \le \tau\} \subseteq C & \Rightarrow & (m \in \tau') \in C \\
(M2) & \{m \in (\tau, M)\} \subseteq C, m \notin \mathsf{Upper}_C(M) & \Rightarrow & (m \in \tau) \in C \\
(M3) & \{m \in (\varsigma_1, \varsigma_2)\} \subseteq C & \Rightarrow & \{m \in \varsigma_1, m \in \varsigma_2\} \subseteq C \\
(M4) & \{m \in \varsigma \leftarrow (m', t)\} \subseteq C, m \ne m' & \Rightarrow & (m \in \varsigma) \in C
\end{array}
$$

Figure 4. Definition of a closed constraint set C, where meta variable w ranges over num, bool, $\overline{(t, M)} \to \tau$, and $[m^\psi : t]$, $\psi \in \{+, -\}$.

where $\psi \in \{+, -\}$). In rule (C3), we write $\overline{t' \le t}$ for $t'_i \le t_i$ and $\overline{\mathcal{M} \subseteq M}$ for $\mathcal{M}_i \subseteq M_i$, where $i \in \{1, 2\}$.

Rule (C4) generates constraints for reading the member m of a pro-type ς. The rule uses a function $\mathsf{field}_C(\varsigma, m)$ (defined below) that returns all possible types of the member m in ς. If a pro-type ς has the member m added indirectly through function/method calls, then $\varsigma \le [m^- : t] \in C$ for some type variable t. The function $\mathsf{fld}_C(\varsigma, m)$ returns the types of member m directly added to ς.

$$
\begin{array}{rcl}
\mathsf{field}_C(\varsigma, m) & = & \{t \mid \varsigma \le [m^- : t] \in C\} \cup \mathsf{fld}_C(\varsigma, m) \\
\mathsf{fld}_C(\varsigma \leftarrow (m, t), m) & = & \{t\} \\
\mathsf{fld}_C(\varsigma \leftarrow (m', t), m) & = & \mathsf{fld}_C(\varsigma, m) \text{ where } m' \ne m \\
\mathsf{fld}_C((\varsigma, M), m) & = & \mathsf{fld}_C(\varsigma, m) \\
\mathsf{fld}_C((\varsigma, \varsigma'), m) & = & \mathsf{fld}_C(\varsigma, m) \cup \mathsf{fld}_C(\varsigma', m) \\
\mathsf{fld}_C(\gamma, m) & = & \mathsf{fld}_C(\varsigma, m), \text{ where } \varsigma \le \gamma \in C \\
\mathsf{fld}_C(\beta, m) & = & \emptyset
\end{array}
$$

Rule (C5) propagates the variable type upper bound of a pro-type ς to all the pro-types derived from ς. The rule uses a relation $\varsigma\ \mathsf{derive}_C\ \varsigma'$ defined for each ς' derived from ς:

$$
\frac{\varsigma' = (\varsigma, M) \mid \varsigma \leftarrow (m, t) \mid (\varsigma, \_) \mid (\_, \varsigma)}{\varsigma\ \mathsf{derive}_C\ \varsigma'} \qquad \frac{\varsigma \le \gamma \in C}{\varsigma\ \mathsf{derive}_C\ \gamma}
$$

Note that this $\varsigma\ \mathsf{derive}_C\ \varsigma'$ relation is ambiguous since ς' can be formed arbitrarily. However, for our purpose, we restrict ς' to the pro-types that actually exist in the type environment.

To see why this rule is needed, consider the example below. Suppose that y at line 4 has type γ, where $\{\beta \leftarrow (m, t_1) \le \gamma, \mathsf{num} \le t_1\} \subseteq C$, and y_1 has the type t, where $\gamma \le t \in C$.

```
1  function F() { this.m = 1; }
2  function f(x) { return x; }
3
4  y = new F();
5  y1 = f(y);
6  if (b) y.m = true;
7  z = y1.m
```

If b is true, then y has a new type $\gamma \leftarrow (m, t_2)$ after line 6, where $\mathsf{bool} \le t_2 \in C$. After line 7, z has the type t', where $t \le [m^+ : t'] \in C$.

The variable z at line 7 may be an integer or boolean. If we only have $\gamma \le t \in C$, then Rule (C4) will only add $\mathsf{num} \le t'$ to C and allow z be used as a number. However, since Rule (C5) adds $\gamma \leftarrow (m, t_2) \le t$ to C, by Rule (C4), we have $\mathsf{bool} \le t' \in C$, which indicates that z may also be a boolean.

Rule (M1) to (M4) propagate member constraint based on subtyping relations and the structure of pro-types. If a constraint of the form $m \in \beta$ is added by the closure rules, then the constraint set is inconsistent since β variable represents empty objects. In Rule (M2), we use a function $\mathsf{Upper}_C(M)$ to obtain the upper bound of M. Since M may contain set variable $\mathcal{M}$, which may be unbounded within a particular constraint set, we let $\mathsf{Upper}^0_C(\mathcal{M}) = T$ for each $\mathcal{M}$, where T represents the maximum set of member names. The upper bound of each M is then calculated iteratively until they converge.

$$
\begin{array}{c}
\mathsf{Upper}^{i+1}_C(M \cup M') = \mathsf{Upper}^i_C(M) \cup \mathsf{Upper}^i_C(M') \\
\mathsf{Upper}^{i+1}_C(M \cap M') = \mathsf{Upper}^i_C(M) \cap \mathsf{Upper}^i_C(M') \\
\mathsf{Upper}^i_C(\emptyset) = \emptyset \quad \mathsf{Upper}^i_C(\{m\}) = \{m\} \\
\mathsf{Upper}^{i+1}_C(\mathcal{M}) = \bigcap_{\mathcal{M} \subseteq M \in C} \mathsf{Upper}^i_C(M) \\
\mathsf{Upper}_C(M) = \mathsf{Upper}^i_C(M) \text{ if } \mathsf{Upper}^{i+1}_C(M) = \mathsf{Upper}^i_C(M)\ \forall M
\end{array}
$$

By Rule (M2), if a member m is selected from a variable of type (τ, M) and m is not in M, then m must be defined in τ, where τ may be a pro-type or obj-type.

4.4 Constraint consistency

We define consistency rules to verify that a closed constraints set does not contain immediate contradictions.

Constraints of the following form are immediately inconsistent:

1. $\tau \le \tau'$ where
   (a) τ is a base type and τ' is a different base type, or a function type or a pro-type,
   (b) τ is a function type and τ' is a base type or a pro-type,
   (c) τ is a pro-type and τ' is a base type or a function type;
2. $\tau \le [m^\psi : t]$ where τ is a base type or function type;
3. $\mathsf{null} \le \tau$, where τ is a base type or function type;
4. $m \in \beta$.

A constraint set C is consistent if no constraint in its closure $\mathsf{cls}(C)$ is immediately inconsistent.

The first two rules check constraints that are obviously incompatible. The third rule makes sure variables of base type or function type cannot be null. It is possible that variables of object types have null value. The last rule represents the error of reading undefined member. If a member m is read on an object that has no definition of m, then the closure rule will eventually generate a constraint of the form $m \in \beta$, where β corresponds to the initial type of the self pointer of the object's constructor.

$$\frac{\forall i \in 1..n.\ \Gamma \vdash Fn_i \quad \Gamma, \emptyset \vdash s \parallel \Gamma', C}{\Gamma \vdash Fn_i^{i \in 1..n}\ s \parallel \Gamma', C} \quad \text{T-Prog}$$

$$\frac{\begin{array}{c}\Gamma_f, \emptyset \vdash s \parallel \Gamma', C \quad \Gamma(f) = \tau_f \setminus C \\ \Gamma_f = \Gamma[\texttt{this} \mapsto a, x \mapsto a_x, a \mapsto (t, \emptyset), a_x \mapsto (t_x, \emptyset)] \\ \tau_f = \Gamma'(a) \times \Gamma'(a_x) \to \Gamma'(\Gamma'(z))\end{array}}{\Gamma \vdash \mathsf{function}\ f(x)\{s; \mathsf{return}\ z\}} \quad \text{T-Fn}$$

$$\frac{\begin{array}{c}\Gamma_F, \emptyset \vdash s \parallel \Gamma', C \quad \Gamma(F) = \tau_F \setminus C \\ \Gamma_F = \Gamma[\texttt{this} \mapsto a, x \mapsto a_x, a \mapsto \beta, a_x \mapsto (t, \emptyset)] \\ \tau_F = \Gamma'(a_x) \to \Gamma'(a)\end{array}}{\Gamma \vdash \mathsf{function}\ F(x)\{s\}} \quad \text{T-Ctr}$$

Figure 5. Typing rules for program, constructor, and function

4.5 Type rules for functions and constructors

The typing rules for functions and constructors are given in Figure 5 and the rules for statements are in Figure 6. The typing rules use the following notations.

A type environment Γ is a mapping from function/constructor names to rc types and a mapping from variables to type aliases, and from type aliases to types. We use type alias to track a variable's type until the variable is assigned with a different type. A type alias a is uniquely mapped to a type while several variables may be mapped to the same type alias.

For any name/variable/alias in the domain of Γ, we define

$$\frac{\Gamma = [\ldots f \mapsto \kappa \ldots]}{\Gamma(f) = \kappa} \quad \frac{\Gamma = [\ldots F \mapsto \kappa \ldots]}{\Gamma(F) = \kappa}$$

$$\frac{\Gamma = [\ldots y \mapsto a \ldots]}{\Gamma(y) = a} \quad \frac{\Gamma = [\ldots a \mapsto \tau \ldots]}{\Gamma(a) = \tau}$$

Given an initial environment Γ with the mapping of functions/-constructors to their types, a program $Fn_i^{i \in 1..n}\ s$ is well-typed if each of the function/constructor definition Fn_i is well-typed and the main program s is well-typed.

The definition of a function f is well-typed given Γ if we can construct a new environment from Γ for the function body so that it is well-typed. In particular, the parameters are assigned extended types with empty member set such as $(t, \emptyset)$. After the function body s is analyzed, the type of the parameter may be extended to (t, M). We use (t, M) as the type of the parameter in the function type. M is used to record the members added to the parameter in s. Though type alias is not needed for the self pointer since it is immutable, the use of type alias a_x in Rule (T-Fn) allows us to keep track of the extension to the parameter type t_x in case it is assigned a different value in the function body.

Given a type environment Γ and a constraint set C, the typing rules for statements derive a possibly new environment Γ' and constraint set C' for a statement s, written as $\Gamma, C \vdash s \parallel \Gamma', C'$. Each rule implicitly enforces the consistency of the constraint set C'. In Rule (T-Fn), the constraint set derived from the body of a function f is the one in the rc type of f defined in the initial type environment (e.g. $\Gamma(f) = \tau_f \setminus C$). This does present a problem for mutually recursive functions and we will address this in Section 5, where we also show the steps to simplify the constraint set.

4.6 Type rules for statements

Typing rules for statements are shown in Figure 6, where we use the following definition of $\tau \cup M$ to represent type extensions.

$$t \cup M \equiv (t, M)$$
$$(t, M) \cup M' \equiv (t, M \cup M')$$
$$\varsigma \cup \mathcal{M} \equiv (\varsigma, \mathcal{M})$$

For an assignment $x = v$, Rule (T-Assn) finds the type of v through Rule (T-Val) with the judgment $\Gamma \vdash v : \tau \setminus C$.

For each appearance of a function f, we can instantiate the rc type $\tau \setminus C$ of f by renaming the variables (including type and set variables) in τ and C. For a recursively defined function, the renaming of its type within itself is the identity function.

To illustrate the typing rules, we show how the function `leftMost` in Figure 1 can be typed as follows:

$$\begin{array}{ll}\Gamma \vdash & \mathsf{function}\ \texttt{leftMost}(z_1, z_2)\{ \\ & \quad x_1 = z_1.\texttt{getX}(); \\ & \quad x_2 = z_2.\texttt{getX}(); \\ & \quad b = x_1 \le x_2 \\ & \quad \texttt{if}(b)\ \{r = z_1;\}\ \texttt{else}\ \{r = z_2;\} \\ & \quad \mathsf{return}\ r; \\ & \}\end{array}$$

where $\Gamma = \{\texttt{leftMost} \mapsto \kappa\}$ and

$$\kappa = t_1 \times t_2 \to t \setminus \begin{array}{l}\{t_1 \le [\texttt{getX}^+ : t_3], t_2 \le [\texttt{getX}^+ : t_4], \\ \texttt{getX} \in t_1, \texttt{getX} \in t_2, \\ t_3 \le t_5 \to t_7, t_4 \le t_6 \to t_8, \\ t_1 \le t_3, t_2 \le t_4, t_7 \le \mathsf{num}, t_8 \le \mathsf{num}, \\ t_1 \le t, t_2 \le t\}.\end{array}$$

By Rule (T-Fn), we construct the type environment Γ_1 to check the function body. so that

$$\Gamma_1 = \Gamma \cup \{z_1 \mapsto a_1, a_1 \mapsto t_1, z_2 \mapsto a_2, a_2 \mapsto t_2\}.$$

Here we omit the type for self pointer since it is never used and we also omit the member set associated with parameter types since they are not extended. From this environment and an empty constraint set, we can apply statement typing rules to each statement in the function body sequentially to derive the final constraint set for the function.

For the first statement, based on Rule (T-Invk), we have

$$\Gamma_1, \emptyset \vdash x_1 = z_1.\texttt{getX}() \parallel \Gamma_2, C_2, \text{ where}$$

$$\begin{array}{l}\Gamma_2 = \Gamma_1[x_1 \mapsto a_1', a_1' \mapsto t_7] \\ C_2 = \{t_1 \le [\texttt{getX}^+ : t_3], \texttt{getX} \in t_1, \\ \qquad\qquad t_3 \le t_5 \to t_7, t_1 \le t_5\}.\end{array}$$

In $t_5 \to t_7$, the type t_5 corresponds to the receiver type of the call `z1.getX()` and the parameter type is omitted since `getX` has no parameter.

Similarly, for the second statement, we have

$$\Gamma_2, C_2 \vdash x_2 = z_2.\texttt{getX}() \parallel \Gamma_3, C_3, \text{ where}$$

$$\begin{array}{l}\Gamma_3 = \Gamma_2[x_2 \mapsto a_2', a_2' \mapsto t_8] \\ C_3 = C_2 \cup \{t_2 \le [\texttt{getX}^+ : t_4], \texttt{getX} \in t_2, \\ \qquad\qquad t_4 \le t_6 \to t_8, t_2 \le t_6\}.\end{array}$$

For the third statement, we don't have a rule to check comparison expression but it can be easily added:

$$\Gamma \vdash x \le y : \mathsf{bool} \setminus \{\Gamma(\Gamma(x)) \le \mathsf{num}, \Gamma(\Gamma(y)) \le \mathsf{num}\}$$

$$\Gamma \vdash n : \mathsf{num} \setminus \emptyset \qquad \Gamma \vdash b : \mathsf{bool} \setminus \emptyset \qquad \Gamma \vdash \mathsf{null} : \mathsf{null} \setminus \emptyset$$

$$\frac{\hat{f} = f \text{ or } \hat{f} = F \quad \Gamma(\hat{f}) = \tau \setminus C \quad \Psi \text{ is a renaming of the type variables in } C}{\Gamma \vdash \hat{f} : \Psi\tau \setminus \Psi C} \quad \text{T-Val}$$

$$\frac{\Gamma \vdash v : \tau \setminus C'}{\Gamma, C \vdash x = v \;\parallel\; \Gamma[x \mapsto a, a \mapsto \tau], C \cup C'} \quad \text{T-Assn}$$

$$\Gamma, C \vdash x = y \;\parallel\; \Gamma[x \mapsto \Gamma(y)], C \quad \text{T-Assn2}$$

$$\frac{\Gamma(\Gamma(y')) = \tau' \quad \Gamma(y) = a_y \quad \Gamma(a_y) = \tau \quad \tau \neq \varsigma \quad C' = C \cup \{\tau \leq [m^- : t], \tau' \leq t\}}{\Gamma, C \vdash y.m = y' \;\parallel\; \Gamma[a_y \mapsto \tau \cup \{m\}], C'} \quad \text{T-Upd}$$

$$\frac{\Gamma(\Gamma(y')) = \tau' \quad \Gamma(y) = a_y \quad \Gamma(a_y) = \varsigma \quad \varsigma' = \varsigma \leftarrow (m, t)}{\Gamma, C \vdash y.m = y' \;\parallel\; \Gamma[a_y \mapsto \varsigma'], C \cup \{\tau' \leq t\}} \quad \text{T-Upd2}$$

$$\frac{\Gamma \vdash F : \tau_F \setminus C_F \quad \Gamma(y) = a_y \quad \Gamma(a_y) = \tau_y \quad C' = \{\tau_F \leq (t, \mathcal{M}) \to \gamma, \tau_y \leq t\}}{\Gamma, C \vdash x = \mathsf{new}\ F(y) \;\parallel\; \Gamma[x \mapsto a, a \mapsto \gamma, a_y \mapsto \tau_y \cup \mathcal{M}], C \cup C' \cup C_F} \quad \text{T-New}$$

$$\frac{\Gamma(\Gamma(y)) = \tau \quad C' = \{\tau \leq [m^+ : t], m \in \tau\}}{\Gamma, C \vdash x = y.m \;\parallel\; \Gamma[x \mapsto a, a \mapsto t], C \cup C'} \quad \text{T-Sel}$$

$$\frac{\begin{array}{c}\Gamma(y) = a \quad \Gamma(y') = a' \quad \Gamma(a) = \tau \quad \Gamma(a') = \tau' \quad \Gamma' = \Gamma[x \mapsto a_x, a_x \mapsto t_x, a \mapsto \tau \cup \mathcal{M}, a' \mapsto \tau' \cup \mathcal{M}'] \\ C' = \{\tau \leq [m^+ : t], t \leq (t_y, \mathcal{M}) \times (t'_y, \mathcal{M}') \to t_x, m \in \tau, \tau \leq t_y, \tau' \leq t'_y\}\end{array}}{\Gamma, C \vdash x = y.m(y') \;\parallel\; \Gamma', C \cup C'} \quad \text{T-Invk}$$

$$\frac{\Gamma, C \vdash s \;\parallel\; \Gamma', C' \quad \Gamma', C' \vdash s' \;\parallel\; \Gamma'', C''}{\Gamma, C \vdash s; s' \;\parallel\; \Gamma'', C''} \quad \text{T-Seq}$$

$$\frac{\Gamma, C \vdash s_1 \;\parallel\; \Gamma_1, C_1 \qquad \Gamma, C \vdash s_2 \;\parallel\; \Gamma_2, C_2 \qquad \mathtt{mrg}(\Gamma_1, \Gamma_2, \Gamma', C')}{\Gamma, C \vdash \mathtt{if}(x)\ \{s_1\}\ \mathtt{else}\{s_2\} \;\parallel\; \Gamma', C_1 \cup C_2 \cup C' \cup \{\Gamma(\Gamma(x)) \leq \mathsf{bool}\}} \quad \text{T-If}$$

Figure 6. Type rules for statements, where for each judgment $\Gamma, C \vdash s \;\parallel\; \Gamma', C'$, it is implicit that C' is consistent.

Thus, we have

$$\Gamma_3, C_3 \vdash b = x_1 \leq x_2 \;\parallel\; \Gamma_4, C_4, \text{ where}$$

$$\Gamma_4 = \Gamma_3[b \mapsto \mathsf{bool}]$$
$$C_4 = C_3 \cup \{t_7 \leq \mathsf{num}, t_8 \leq \mathsf{num}\}.$$

For the fourth statement, we apply Rule (T-If) so that

$$\Gamma_4, C_4 \vdash \mathtt{if}(b)\ \{r = z_1;\}\ \mathtt{else}\ \{r = z_2;\} \;\parallel\; \Gamma_5, C_5, \text{ where}$$

$$\Gamma_5 = \Gamma_4[r \mapsto a_r, a_r \mapsto t]$$
$$C_5 = C_4 \cup \{t_1 \leq t, t_2 \leq t\}.$$

The redundant constraint $\mathsf{bool} \leq \mathsf{bool}$ is omitted.

Rule (T-If) uses a predicate $\mathtt{mrg}(\Gamma_t, \Gamma_f, \Gamma', C')$ that merges the type environments of both branches Γ_t and Γ_f into a new environment Γ' and a constraint set C'. By Rule (T-Assn2), the assignment in each branch of the if statement creates a new environment

$$\begin{array}{rcl}\Gamma_t & = & \Gamma_4[r \mapsto a_t, a_t \mapsto t_1], \\ \Gamma_f & = & \Gamma_4[r \mapsto a_f, a_f \mapsto t_2].\end{array}$$

When we merge the two environments, we get

$$\Gamma_4[r \mapsto a_r, a_r \mapsto t] \text{ and } \{t_1 \leq t, t_2 \leq t\}.$$

Below is the definition of the merge predicates. The merging process includes the merging of mappings from variables to type aliases and the merging of mappings from type aliases to types.

$$\mathtt{mrg}(\Gamma, \Gamma, \Gamma, \emptyset) \qquad \frac{a \notin \mathsf{dom}(\Gamma_2) \quad \mathtt{mrg}(\Gamma_1, \Gamma_2, \Gamma, C)}{\mathtt{mrg}((a \mapsto \tau, \Gamma_1), \Gamma_2, \Gamma, C)}$$

$$\frac{\mathtt{mrg}(\Gamma_1, \Gamma_2, \Gamma, C)}{\mathtt{mrg}(\Gamma_2, \Gamma_1, \Gamma, C)} \qquad \frac{y \notin \mathsf{dom}(\Gamma_2) \quad \mathtt{mrg}(\Gamma_1, \Gamma_2, \Gamma, C)}{\mathtt{mrg}((y \mapsto a, \Gamma_1), \Gamma_2, \Gamma, C)}$$

$$\frac{\varsigma_1 \neq \varsigma_2 \quad \mathtt{mrg}(\Gamma_1, \Gamma_2, \Gamma, C)}{\mathtt{mrg}((a \mapsto \varsigma_1, \Gamma_1), (a \mapsto \varsigma_2, \Gamma_2), (a \mapsto (\varsigma_1, \varsigma_2), \Gamma), C)}$$

$$\frac{\mathtt{mrg}(\Gamma_1, \Gamma_2, \Gamma, C)}{\mathtt{mrg}((a \mapsto t, \Gamma_1), (a \mapsto (t, M), \Gamma_2), (a \mapsto t, \Gamma), C)}$$

$$\frac{\mathtt{mrg}(\Gamma_1, \Gamma_2, \Gamma, C) \quad \tau = (t, M_1 \cap M_2)}{\mathtt{mrg}((a \mapsto (t, M_1), \Gamma_1), (a \mapsto (t, M_2), \Gamma_2), (a \mapsto \tau, \Gamma), C)}$$

$$\frac{\begin{array}{c}\Gamma_1(a_1) = \tau_1 \quad \Gamma_2(a_2) = \tau_2 \quad a_1 \neq a_2 \\ \mathtt{mrg}(\Gamma_1, \Gamma_2, \Gamma, C) \quad C' = C \cup \{\tau_1 \leq t, \tau_2 \leq t\}\end{array}}{\mathtt{mrg}((y \mapsto a_1, \Gamma_1), (y \mapsto a_2, \Gamma_2), (y \mapsto a, a \mapsto t, \Gamma), C')}$$

As another example, consider the constructor `Line` in Figure 2 with some simplification. We can derive its type as follows:

$$\Gamma \vdash \quad \begin{array}{l}\mathsf{function}\ \mathsf{Line}(\mathtt{p1},\mathtt{p2})\{\\ \quad \mathtt{this.p1} = \mathtt{p1};\\ \quad \mathtt{p1.geom} = \mathtt{this};\\ \quad \mathtt{this.p2} = \mathtt{p2};\\ \quad \mathtt{p2.geom} = \mathtt{this};\\ \}\end{array}$$

where $\Gamma = \{\mathsf{Line} \mapsto \kappa\}$,

$$\kappa = (t_1, \{\mathtt{geom}\}) \times (t_2, \{\mathtt{geom}\}) \to \varsigma_2 \setminus \begin{array}{r}\{\ t_1 \le [\mathtt{geom}^- : t_{\mathtt{g1}}],\\ t_2 \le [\mathtt{geom}^- : t_{\mathtt{g2}}],\\ t_1 \le t_{\mathtt{p1}}, t_2 \le t_{\mathtt{p2}},\\ \varsigma_1 \le t_{\mathtt{g1}}, \varsigma_2 \le t_{\mathtt{g2}}\ \},\end{array}$$

$\varsigma_1 = \beta \leftarrow (\mathtt{p1}, t_{\mathtt{p1}})$, and $\varsigma_2 = \varsigma_1 \leftarrow (\mathtt{p2}, t_{\mathtt{p2}})$.

By Rule (T-Ctr), we create a type environment Γ_1 for the function body so that

$$\begin{array}{rcl}\Gamma_1 & = & \Gamma[\mathtt{this} \mapsto a, a \mapsto \beta,\\ & & \mathtt{p1} \mapsto a_1, a_1 \mapsto (t_1, \emptyset),\\ & & \mathtt{p2} \mapsto a_2, a_2 \mapsto (t_2, \emptyset)].\end{array}$$

Note the typing rule assumed one parameter but the same design applies to multiple parameters.

For the first statement, we apply Rule (T-Upd2) so that

$$\Gamma_1, \emptyset \vdash \mathtt{this.p1} = \mathtt{p1} \ \parallel\ \Gamma_2, C_2 \text{ where}$$

$$\begin{array}{l}\Gamma_2 = \Gamma_1[a \mapsto \varsigma_1],\\ C_2 = \{(t_1, \emptyset) \le t_{\mathtt{p1}}\}.\end{array}$$

For simplicity, we replace $(t_1, \emptyset) \le t_{\mathtt{p1}}$ with $t_1 \le t_{\mathtt{p1}}$ so that $C_2 = \{t_1 \le t_{\mathtt{p1}}\}$ since they have the same effect.

For the second statement, we apply Rule (T-Upd) so that

$$\Gamma_2, C_2 \vdash \mathtt{p1.geom} = \mathtt{this} \ \parallel\ \Gamma_3, C_3 \text{ where}$$

$$\begin{array}{l}\Gamma_3 = \Gamma_2[a \mapsto (t_1, \{\mathtt{geom}\})]\\ C_3 = C_2 \cup \{(t_1, \emptyset) \le [\mathtt{geom}^- : t_{\mathtt{g1}}], \varsigma_1 \le t_{\mathtt{g1}}\}.\end{array}$$

Again, we replace $(t_1, \emptyset) \le [\mathtt{geom}^- : t_{\mathtt{g1}}]$ with $t_1 \le [\mathtt{geom}^- : t_{\mathtt{g1}}]$ for simplicity.

The last two statements are typed the same way.

To see how Rule (T-New) is applied, consider the statement `line = new Line(p1, p2)` (line 17 of Figure 2), where we assume `p1` and `p2` are point objects of the type $\varsigma_{\mathtt{p1}}$ and $\varsigma_{\mathtt{p2}}$ respectively. By Rule (T-New),

$$\Gamma, C \vdash \mathtt{line} = \mathsf{new}\ \mathsf{Line}(\mathtt{p1}, \mathtt{p2}) \ \parallel\ \Gamma', C' \text{ where}$$

$$\begin{array}{l}\Gamma' = \Gamma[\mathtt{line} \mapsto a, a \mapsto \gamma,\\ \qquad \mathtt{p1} \mapsto (\varsigma_{\mathtt{p1}}, \mathcal{M}_1), \mathtt{p2} \mapsto (\varsigma_{\mathtt{p2}}, \mathcal{M}_2)],\\ C' = C \cup C_{\mathsf{Line}} \cup\\ \qquad \{\tau_{\mathsf{Line}} \le (t_1, \mathcal{M}_1) \times (t_2, \mathcal{M}_2) \to \gamma, \varsigma_{\mathtt{p1}} \le t_1, \varsigma_{\mathtt{p2}} \le t_2\},\end{array}$$

and $\tau_{\mathsf{Line}} \setminus C_{\mathsf{Line}}$ is an instance of the rc type of the Line constructor from the judgment $\Gamma \vdash \mathsf{Line} : \tau_{\mathsf{Line}} \setminus C_{\mathsf{Line}}$.

If we consider the closure of C', then it contains the following constraints $C_{\mathtt{p1}}$ related to the type of `p1` in Γ':

$$\{\varsigma_{\mathtt{p1}} \le t_1, t_1 \le t_1', t_1' \le [\mathtt{geom}^- : t_{\mathtt{g1}}'], \varsigma_1' \le t_{\mathtt{g1}}', \mathcal{M}_1 \subseteq \{\mathtt{geom}\}\}$$

The primed variables are unique instances of the variables in the rc type κ of the Line function through renaming.

The typing of the statement `g = p1.geom` (line 20 of Figure 2) generates the environment

$$\Gamma'' = \Gamma'[\mathtt{g} \mapsto t]$$

and the constraints

$$C'' = C' \cup \{(\varsigma_{\mathtt{p1}}, \mathcal{M}_1) \le [\mathtt{geom}^+ : t], \mathtt{geom} \in (\varsigma_{\mathtt{p1}}, \mathcal{M}_1)\}.$$

We can see that C'' is consistent since $\mathsf{Upper}_{C''}(\mathcal{M}_1) = \{\mathtt{geom}\}$ and the closure rule (M2) does not reduce the constraint $\mathtt{geom} \in (\varsigma_{\mathtt{p1}}, \mathcal{M}_1)$ to $\mathtt{geom} \in \varsigma_{\mathtt{p1}}$. Also, by closure rule (C4), the closure of C'' has the constraint $\varsigma_1' \le t$. Note that ς_1' is a pro-type of the line object and the variable `g` is an alias of `line`. By closure rule (C5), the closure of C'' also contains $\varsigma_2' \le t$ so that the type of the variable `g` reflects the latest state of the line object.

In summary, our typing rules are flexible enough to handle the polymorphic types and object extensions. In the appendix, we give an operational semantics for our language and show that if a program is typable then it will not access undefined object members.

5. Type Inference

The typing rules for statements are sufficient for type inference though the typing rules for functions and constructors do not consider mutually recursive definitions. Also, we need to simplify the type constraint set for each inferred rc type to reduce its size.

Each time a function is used, we instantiate its rc type by Rule (T-Val), which introduces fresh variables to the constraint set, but this can cause the constraint sets inferred from mutually recursive functions to have unbounded size. To solve this problem, we adopt a simple strategy that mutually recursive functions have monomorphic types when they are used among themselves, while they have polymorphic types elsewhere. When we apply Rule (T-Val) for monomorphic function types, we can let the renaming Ψ be identity function. While this may be restrictive for recursively defined functions, as a future work, we hope to find evidence that this is acceptable for practical programs.

$$\frac{\begin{array}{c}\Gamma_1 = \emptyset\\ \vdash_{\mathsf{inf}} Fn_i^{\,i \in \bigcup_{j \in 1..k-1} S_j} \ \parallel\ \Gamma_k \text{ if } k > 1\\ \Gamma_k \vdash_{\mathsf{inf}} Fn_i^{\,i \in S_k} \ \parallel\ \Gamma_{k+1}\end{array}}{\vdash_{\mathsf{inf}} Fn_i^{\,i \in \bigcup_{j \in 1..k} S_j} \ \parallel\ \Gamma_{k+1}} \quad \text{I-Prog}$$

$$\frac{\begin{array}{c}\forall i \in S_k.\ \hat{f}_i \text{ is the name of } Fn_i\\ \Gamma_k[\hat{f}_i \mapsto t_i^{\,i \in S_k}] \vdash_{\mathsf{inf}} Fn_i : \tau_i \setminus C_i\\ C = \bigcup_i C_i \quad C' = C[\tau_i / t_i^{\,i \in S_k}] \quad C' \text{ is consistent}\\ \Gamma_{k+1} = \Gamma_k[\hat{f}_i \mapsto \tau_i \setminus \mathsf{reach}^+_{C'}(\tau_i)^{i \in S_k}]\end{array}}{\Gamma_k \vdash_{\mathsf{inf}} Fn_i^{\,i \in S_k} \ \parallel\ \Gamma_{k+1}} \quad \text{I-Prog2}$$

$$\frac{\begin{array}{c}\Gamma, \emptyset \vdash s \ \parallel\ \Gamma', C\\ \Gamma = \Gamma_f[\mathtt{this} \mapsto a, x \mapsto a_x,\\ \qquad a \mapsto (t, \emptyset), a_x \mapsto (t_x, \emptyset)]\\ \tau_f = \Gamma'(a) \times \Gamma'(a_x) \to \Gamma'(\Gamma'(z))\end{array}}{\Gamma_f \vdash_{\mathsf{inf}} \mathsf{function}\ f(x)\{s; \mathsf{return}\ z\} : \tau_f \setminus C} \quad \text{I-Fn}$$

$$\frac{\begin{array}{c}\Gamma, \emptyset \vdash s \ \parallel\ \Gamma', C\\ \Gamma = \Gamma_F[\mathtt{this} \mapsto a, x \mapsto a_x, a \mapsto \beta, a_x \mapsto (t, \emptyset)]\\ \tau_F = \Gamma'(a_x) \to \Gamma'(a)\end{array}}{\Gamma_F \vdash_{\mathsf{inf}} \mathsf{function}\ F(x)\{s\} : \tau_F \setminus C} \quad \text{I-Ctr}$$

Figure 7. Inference rules for functions

The inference rules for functions and constructors are shown in Figure 7. Informally, the type inference algorithm has three steps:

1. Given a program $Fn_i^{i\in 1..n}; s$, create a dependency graph (V, E) where $V = \{i \mid i \in 1..n\}$ and $(i, j) \in E$ if Fn_i depends on Fn_j. Find the strongly connected components S_k, $k \in 1..K$ of the graph so that if S_j depends on S_i then $j > i$.
2. Let Γ_1 be empty set. For $k > 1$, assume we have inferred types for each definition in S_j where $j = 1..k-1$ to have Γ_k.

   For each S_k, apply inference rules to each function/constructor Fn_i where $i \in S_k$ so that $\Gamma_k \vdash_{\mathsf{inf}} Fn_i : \tau_i \setminus C_i$ (Rule (I-Prog2) in Figure 7), where Γ_i maps the function/constructor in S_j, $j \in 1..k-1$ to inferred types and maps each Fn_i in S_k to an unconstrained type variable t_i. The type inference rules for statements are the same as typing rules for statements with the following additions to extract the variable type of function/constructor in S_k.

$$\frac{\Gamma(f) = t}{\Gamma \vdash f : t} \quad \frac{\Gamma(F) = t}{\Gamma \vdash F : t}$$

   After obtaining the type constraints C_i, we check the consistency of $C = \bigcup_{i\in S_k} C_i$ and simplify the type constraints for each function/constructor so that Fn_i has the type of $\tau_i \setminus \mathsf{reach}^+_{C'}(\tau_i)$, where C' is C with each type variable t_i of function/constructor in S_k replaced by its inferred type τ_i.

   The definition of $\mathsf{reach}^+_C(\tau)$ is shown in Figure 8.
3. After the types of all functions and constructors are inferred, we obtain a type environment Γ_K and then check the main program s with $\Gamma_K, \emptyset \vdash s \parallel \Gamma, C$, where C is consistent.

Each time a function of the type $\tau_f \setminus C_f$ is used in a context with the constraint set C, we add C_f to C. However, not all constraints in C_f is needed since we are only interested in those that are related to the type τ_f. Specifically, τ_f may only appear in a constraint of the form $\tau_f \le \tau$ in C (Rule (T-Upd), (T-New), and (T-Invk)). Thus, we define $\mathsf{reach}^+_{C_f}(\tau_f)$ to select the constraints in C_f that are relevant to the closure of C related to $\tau_f \le \tau$.

Informally, $\mathsf{reach}^+_C(\tau)$ finds type constraints related to the lower bounds of τ in C, while $\mathsf{reach}^-_C(\tau)$ selects constraints related to the upper bounds. Redundant constraints such as $\varsigma \leftarrow (m, t) \le [m^+ : t']$ are not included since they can be reduced to more basic constraints after closure operations. $\mathsf{reach}^+_C(\varsigma)$ keeps constraints in C related to ς itself. $\mathsf{reach}^+_C(\mathcal{M})$ includes constraints of the form $m \in (\tau, M)$ in C where $\mathcal{M}$ affects the upper bound of M. $\mathsf{reach}^-_C(M)$ has constraints in C related to the upper bound of M.

The simplification of the rc types does not affect the functions within a strong connected component since they have monomorphic types among themselves and their constraints are pooled together to check consistency (Rule (I-Prog2)). When the functions are used in other context, we need to make sure the simplification does not remove necessary constraints. That is, we want to show that if

$$\Gamma[\ldots f \mapsto \tau_f \setminus \mathsf{reach}^+_{C_f}(\tau_f) \ldots], C \vdash s \parallel \Gamma', C',$$

then

$$\Gamma[\ldots f \mapsto \tau_f \setminus C_f \ldots], C \vdash s \parallel \Gamma'', C'',$$

where C' and C'' are consistent. Since each time f is used, its type is instantiated by a renaming Ψ, the resulting constraint set ΨC_f has no overlap with C. When we add ΨC_f to C, they have no interaction until a constraint of the form $\Psi\tau_f \le \tau$ is added by the statement typing rules. We need to verify that if

$$\Psi\mathsf{reach}^+_{C_f}(\tau_f) \cup \{\Psi\tau_f \le \tau\} \cup C$$

is consistent, then so is

$$\Psi C_f \cup \{\Psi\tau_f \le \tau\} \cup C.$$

Since Ψ only changes the names of variables, $\Psi\mathsf{reach}^+_{C_f}(\tau_f) = \mathsf{reach}^+_{\Psi C_f}(\Psi\tau_f)$. Thus, we need the following lemma.

Lemma 5.1. *Assume C_f and C are consistent and they do not have variables in common. Also, the variables in τ do not appear in C_f. If $C \cup \{\tau_f \le \tau\} \cup \mathsf{reach}^+_{C_f}(\tau_f)$ is consistent, then $C \cup \{\tau_f \le \tau\} \cup C_f$ is also consistent.*

Let $C' = C \cup \{\tau_f \le \tau\} \cup C_f$ and $C'' = C \cup \{\tau_f \le \tau\} \cup \mathsf{reach}^+_{C_f}(\tau_f)$. The proof is to show that $\mathsf{cls}(C') \subseteq \mathsf{cls}(C'') \cup \mathsf{cls}(C_f)$.

Since simplification preserves the typability of functions, we can conclude that a program is typable if we can infer its types.

Theorem 5.2. *If $\vdash_{\mathsf{inf}} Fn_i^{i\in 1..n} \parallel \Gamma_K$ and $\Gamma_K, \emptyset \vdash s \parallel \Gamma'_K, C$, then $\exists\Gamma$ such that $\Gamma \vdash Fn_i^{i\in 1..n}$ and $\Gamma, \emptyset \vdash s \parallel \Gamma', C'$.*

6. Related work

Type inference and analysis for scripting languages Anderson et al. [4] developed a type inference system for a small subset of JavaScript that supports explicit member extensions on objects and their type system ensures that the new members may only be accessed after the extensions. Their algorithm only infers monomorphic types with explicit member extension while we allow parametric polymorphism, implicit object extensions, and strong updates and unrestricted extensions to new objects.

Recency types of Heidegger and Thiemann [12] have the similar goal of preventing the access of undefined members through type-based analysis. Their approach uses two kinds of object types: singleton type and summary type, where each singleton type is associated with an abstract location and it has to be demoted to a summary type when the next object is allocated at the same location. Objects of singleton types can receive strong updates and extensions while objects of summary types are fixed. In comparison to our work, recency type system allows singleton types for object members and function parameters. However, it does not support parametric polymorphism and singleton type objects can no longer be extended after they are assigned to variables of summary types. Pro-type objects can still be extended even if they have aliases of obj-types. Also, in their formalism, abstract locations are assigned to new expressions that return empty objects. This may be restrictive for modeling constructor functions since only the most recent object created from the constructor can have singleton type.

We have previously developed a type inference algorithm for a subset of JavaScript with implicit object extensions [25]. That algorithm also uses two kinds of object types: singleton type that allows strong update and arbitrary extensions and obj-type with limited extensions. The main difference is that it does not support parametric polymorphism.

Jensen et al. [13] have implemented a practical analyzer to detect possible runtime errors of JavaScript program. Their approach is based on abstract interpretation and uses recency information. The analyzer can report the absence of errors based on some inputs but it does not infer types. Earlier work of Thiemann [22] proposed a type system for a subset of JavaScript language to detect conversion errors of JavaScript values. The type system models automatic conversions in JavaScript but it does not model recursive or flow sensitive types.

Guha et al. [11] developed a type-checker for JavaScript that combines typing and flow analysis to support the use of local control and state to reason informally about types. Their system uses tags to identify primitives, functions, and locations. They use flow analysis to insert appropriate tag checks at each program point and then apply flow-insensitive type-checker to verify the correctness. They do not support objects or recursive types.

$$
\begin{array}{rcl}
\mathsf{reach}^-_C(t) & = & \bigcup_{t \le w \in \mathsf{cls}(C)} \mathsf{reach}^-_C(w) \cup \{t \le w\} \cup \bigcup_{(m \in t) \in \mathsf{cls}(C)} \{m \in t\} \\
\mathsf{reach}^+_C(t) & = & \bigcup_{\tau \le t \in \mathsf{cls}(C)} \mathsf{reach}^+_C(\tau) \cup \{\tau \le t\} \\
\mathsf{reach}^-_C([m^+ : t]) & = & \mathsf{reach}^-_C(t) \qquad \mathsf{reach}^-_C([m^- : t]) = \mathsf{reach}^+_C(t) \\
\mathsf{reach}^-_C(\overline{(t, \mathcal{M})} \to \tau) & = & \mathsf{reach}^-_C(\tau) \cup \bigcup_i (\mathsf{reach}^+_C(t_i) \cup \mathsf{reach}^+_C(\mathcal{M}_i)) \\
\mathsf{reach}^+_C(\overline{(t, M)} \to \tau) & = & \mathsf{reach}^+_C(\tau) \cup \bigcup_i (\mathsf{reach}^-_C(t_i) \cup \mathsf{reach}^-_C(M_i)) \\
\mathsf{reach}^-_C(\mathsf{num}) & = & \mathsf{reach}^-_C(\mathsf{bool}) = \mathsf{reach}^-_C(\mathsf{null}) = \emptyset \\
\mathsf{reach}^+_C(\mathsf{num}) & = & \mathsf{reach}^+_C(\mathsf{bool}) = \mathsf{reach}^+_C(\mathsf{null}) = \emptyset \\
\\
\mathsf{reach}^+_C((t, M)) & = & \mathsf{reach}^+_C(t) \\
\mathsf{reach}^+_C(\varsigma) & = & \mathsf{reach}_C(\varsigma) \cup \bigcup_{\varsigma \le [m^- : t] \in \mathsf{cls}(C)} (\mathsf{reach}^+_C(t) \cup \{\varsigma \le [m^- : t]\}) \\
\\
\mathsf{reach}_C((\varsigma, M)) & = & \mathsf{reach}_C(\varsigma) \\
\mathsf{reach}_C((\varsigma_1, \varsigma_2)) & = & \mathsf{reach}_C(\varsigma_1) \cup \mathsf{reach}_C(\varsigma_2) \\
\mathsf{reach}_C(\varsigma \leftarrow (m, t)) & = & \mathsf{reach}_C(\varsigma) \cup \mathsf{reach}^+_C(t) \\
\mathsf{reach}_C(\gamma) & = & \mathsf{reach}_C(\varsigma), \varsigma \le \gamma \in \mathsf{cls}(C) \qquad \mathsf{reach}_C(\beta) = \emptyset \\
\\
\mathsf{reach}^-_C(M) & = & \bigcup_{\mathcal{M} \in R_C(M), \mathcal{M} \subseteq M' \in \mathsf{cls}(C)} \{\mathcal{M} \subseteq M'\} \\
\mathsf{reach}^+_C(\mathcal{M}) & = & \bigcup_{\mathcal{M} \in R_C(M), (m \in (\tau, M)) \in \mathsf{cls}(C)} \{m \in (\tau, M)\} \\
R_C(\{m\}) & = & R_C(\emptyset) = \emptyset \\
R_C(M \cup M') & = & R_C(M \cap M') = R_C(M) \cup R_C(M') \\
R_C(\mathcal{M}) & = & \{\mathcal{M}\} \cup \bigcup_{\mathcal{M} \subseteq M \in \mathsf{cls}(C)} R_C(M)
\end{array}
$$

Figure 8. Constraint simplification

Typed Scheme [23] is an explicitly typed extension of Scheme language. The extension introduced a notion of occurrence typing to account for type tests and type predicates so that the control flow information of the program can be used to assign subtypes of a parameter to distinct occurrences.

DRuby [9, 10] is a tool to infer types for Ruby, which is a class-based scripting language. DRuby includes a type system with features such as union, intersection types, object types, self-type, parametric polymorphism, and tuple types. Their type inference is also a constraint-based analysis.

Type inference for class and object-based languages There are a number of studies on type inference for class-based languages. Palsberg et al. [14] have developed a type inference algorithm based on ideas of flow analysis for object-oriented programs with inheritance, assignments, and late binding. The purpose is to guarantee all messages are understood while allowing polymorphic methods. The algorithm handles late binding with conditional constraints and solves the constraints by least fixed-point derivation. Similar algorithm was applied to object-based language SELF [3] that features objects with dynamic inheritance. Plevyak and Chien [18] extended this flow-based approach for better precision via an incremental algorithm. Further enhancement on precision and efficiency were made by Agesen in his Cartesian Product Algorithm [2], which was applied to type inference for Python programs to improve compiled code [20].

The rc type used in this work is adopted from the design of Eifrig et al. [8], who developed a polymorphic, constraint-based type inference algorithm for a class-based language with polymorphic types. Their goal was to mitigate the trade-off between inheritance and subtyping. The rc types are also used in a type inference algorithm for Java [24] to verify the correctness of downcasts. Their inference algorithm extends Agesen's Cartesian Product Algorithm with the ability to analyze data polymorphic programs. In comparison, their algorithms are for class-based languages while this work applies to object-based languages with dynamic extensions.

As for type inference for object-based languages, Palsberg developed efficient type inference algorithms [15] with recursive types and subtyping for Abadi Cardelli object calculus [1], which has method override and subsumption but not object extension. Similar algorithms were developed for inferring object types for an object calculus with covariant read-only fields [17] and supporting record concatenation [16].

Type inference for dynamically typed languages is not scalable to very large programs. Spoon and Shivers [21] have developed a type inference algorithm that trades precision for speed using a demand-driven approach, which solves user provided goals by possibly generating more subgoals. They manage the number of active goals with a subgoal pruning technique, which is to provide a trivially correct answer to a goal to avoid having further subgoals. The balance between precision and scalability may be achieved by choosing pruning thresholds.

Type systems for languages with object extensions Also related is the work of Gianantonio et al. [7] on lambda calculus of objects with self-inflicted extension. They separate the members of an object type into two parts: interface part and reservation part. After an extension, the extended member moves from reservation part to the interface part. They also distinguish two kinds of object types: pro-type and obj-type. A pro-type's reservation part may be extended but no subtyping is allowed on pro-types. A pro-type may be promoted to obj-type which allows covariant subtyping but obj-types' reservation parts may not be extended. Our pro-type objects are similar to theirs except that our pro-type objects do not lose the ability of having strong updates even after it is assigned to parameters or fields of obj-type.

Bono and Fisher [5] proposed an imperative, first-order calculus with object extensions, which also distinguishes extensible pro-types without subtyping from sealed obj-types that allow width and depth subtyping. Their objective is to show that Java-style classes and mixins can be encoded in their calculus through object extensions and encapsulation.

7. Conclusion

We have presented a polymorphic constraint-based type inference algorithm for a small subset of JavaScript. The goal is to prevent accessing an object's member before it is defined. The type system supports explicit and implicit extension to objects and allow strong updates to new objects. Our type inference algorithm is modular so that large programs can be checked incrementally. The inferred rc-types can be simplified so that their constraint sets do not grow with the size of the program. We have implemented a prototype for checking simple JavaScript programs based on the presented algorithm and the source code is available at `http://guangzhou.cs.uwm.edu/javascript`.

Our primary focus is to keep track of member addition/update to objects during and after object initialization, which can be useful for some programs that exhibit this behavior [19]. However, our system is lack of many important features found in real world JavaScript programs such as prototypes, variadic functions, eval function, member deletion, runtime type tests, and objects as associative arrays. For future work, we would like to investigate the addition of prototypes and runtime type tests.

References

[1] M. Abadi and L. Cardelli. *A Theory of Objects*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1996.

[2] O. Agesen. The Cartesian Product Algorithm: Simple and Precise Type Inference Of Parametric Polymorphism. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP'95)*, pages 2–26, 1995.

[3] O. Agesen, J. Palsberg, and M. I. Schwartzbach. Type Inference of SELF. In *Proceedings of the 7th European Conference on Object-Oriented Programming (ECOOP'93)*, pages 247–267, 1993.

[4] C. Anderson, S. Drossopoulou, and P. Giannini. Towards Type Inference for JavaScript. In *19th European Conference on Object-Oriented Programming (ECOOP'05)*, pages 428–452, July 2005.

[5] V. Bono and K. Fisher. An Imperative, First-Order Calculus with Object Extension. In *the 12th European Conference on Object-Oriented Programming (ECOOP'98)*, pages 462–497, 1998.

[6] D. Crockford. *JavaScript: The Good Parts*. O'Reilly Media, 2008.

[7] P. Di Gianantonio, F. Honsell, and L. Liquori. A Lambda Calculus of Objects With Self-Inflicted Extension. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'98)*, pages 166–178, 1998.

[8] J. Eifrig, S. Smith, and V. Trifonov. Sound Polymorphic Type Inference for Objects. *SIGPLAN Not.*, 30(10):169–184, 1995.

[9] M. Furr, J.-h. D. An, and J. S. Foster. Profile guided static typing for dynamic scripting languages. In *Object-Oriented Programming Systems, Languages, and Applications (OOPSLA'09)*, 2009.

[10] M. Furr, J.-h. D. An, J. S. Foster, and M. Hicks. Static type inference for Ruby. In *Proceedings of the 2009 ACM symposium on Applied Computing (SAC'09)*, pages 1859–1866, 2009.

[11] A. Guha, C. Saftoiu, and S. Krishnamurthi. Typing local control and state using flow analysis. In *Proceedings of the 20th European conference on Programming languages and systems*, pages 256–275, 2011.

[12] P. Heidegger and P. Thiemann. Recency Types for Analyzing Scripting Languages. In *Proceedings of the 24th European Conference on Object-Oriented Programming (ECOOP'10),*, pages 200–224, 2010.

[13] S. H. Jensen, A. Møller, and P. Thiemann. Type Analysis for JavaScript. In *16th International Static Analysis Symposium (SAS'09)*, August 2009.

[14] N. Oxhøj, J. Palsberg, and M. I. Schwartzbach. Making Type Inference Practical. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'92)*, pages 329–349, 1992.

[15] J. Palsberg. Efficient Inference of Object Types. *Inf. Comput.*, 123(2): 198–209, 1995.

[16] J. Palsberg and T. Zhao. Type Inference for Record Concatenation and Subtyping. *Inf. Comput.*, 189(1):54–86, 2004.

[17] J. Palsberg, T. Zhao, and T. Jim. Automatic discovery of covariant read-only fields. *ACM Trans. Program. Lang. Syst.*, 27(1):126–162, 2005.

[18] J. Plevyak and A. A. Chien. Precise Concrete Type Inference for Object-Oriented Languages. In *Proceedings of the 9th annual conference on Object-oriented programming systems, language, and applications (OOPSLA'94)*, pages 324–340, 1994.

[19] G. Richards, S. Lesbrene, B. Burg, and J. Vitek. An Analysis of the Dynamic Behavior of JavaScript Programs. In *Proceedings of the ACM Programming Language Design and Implementation Conference (PLDI)*, June 2010.

[20] M. Salib. Faster than C: Static Type Inference with Starkiller. In *in PyCon Proceedings, Washington DC*, pages 2–26, 2004.

[21] S. A. Spoon and O. Shivers. Demand-Driven Type Inference with Subgoal Pruning: Trading Precision for Scalability. In *Proceedings of the 18th European Conference on Object-Oriented Programming, (ECOOP'04)*, pages 51–74, 2004.

[22] P. Thiemann. Towards a Type System for Analyzing JavaScript Programs. In *14th European Symposium on Programming (ESOP'05)*, pages 408–422, 2005.

[23] S. Tobin-Hochstadt and M. Felleisen. The design and implementation of typed scheme. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '08, pages 395–406, 2008.

[24] T. Wang and S. F. Smith. Precise Constraint-Based Type Inference for Java. In *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP'01)*, pages 99–117, 2001.

[25] T. Zhao. Type inference for scripting languages with implicit extension. In *International Workshop on Foundations of Object-Oriented Languages*, 2010.

A. Semantics and Type Soundness

A.1 Operational semantics

We define a big-step semantics for our language in Figure 9. First, we give a few definitions used in the semantics.

A heap H is a mapping from object labels ι to object values o, which maps member names to values. A value v is either an object label, a function name, a primitive value, or null.

$$\begin{array}{lcl} v & ::= & \iota \mid f \mid n \mid b \mid \mathsf{null} \\ o & ::= & \{m_i \mapsto {v_i}^{i \in 1..n}\} \\ H & ::= & \{\iota_i \mapsto {o_i}^{i \in 1..n'}\} \end{array}$$

We can extract the object value from the heap through its label.

$$\frac{H = \{\ldots \iota \mapsto o \ldots\}}{H(\iota) = o}$$

Similarly, we can select a member from an object value through member name if the member is defined in the object.

$$\frac{o = \{\ldots m \mapsto v \ldots\}}{o(m) = v}$$

Otherwise, $o(m) = \mathsf{undef}$, which says m is undefined in o. Note that undef is not the *undefined* property in JavaScript.

We use the symbol χ to represent a stack that maps local variables to to their values and maps a special variable FT to the the declarations of functions and constructors.

$$\chi ::= \{y_i \mapsto {v_i}^{i \in 1..n}, \mathsf{FT} \mapsto {Fn_j}^{j \in 1..n'}\}$$

We can find the value of a name y from the stack if it is in the domain of the stack.

$$\frac{\chi = \{\ldots y \mapsto v \ldots\}}{\chi(y) = v}$$

If y is not defined in the domain of χ, then $\chi(y) = \mathsf{undef}$. Moreover, $\mathsf{lookup}(f, {Fn_i}^{i \in 1..n}) = Fn_j$ if Fn_j is the declaration of the function f and $\mathsf{lookup}(F, {Fn_i}^{i \in 1..n}) = Fn_j$ if Fn_j is the declaration of the constructor F, where $j \in 1..n$.

The reduction of a statement is written in form of $H, \chi, s \rightsquigarrow H', \chi'$, which means that the execution of a statement s given the configuration of a heap H and a stack χ results in a new configuration H', χ'.

The reduction rules are mostly straightforward and they do not consider runtime errors, which will be defined next. A statement s can write to a variable x not defined in χ and after the execution of s, χ is extended with the definition of x. The reduction of *if* statement uses a predicate *trim* to remove variables that are not defined in both branches.

$$\frac{\forall i = 1, 2.\ \chi_i' = \{y \mapsto \chi_i(y) \mid y \in \mathsf{dom}(\chi_1) \cap \mathsf{dom}(\chi_2)\}}{\mathtt{trim}(\chi_1, \chi_2, \chi_1', \chi_2')}$$

A.1.1 Runtime errors

Since big step semantics cannot distinguish a program stuck with runtime error from divergence, we define rules to propagate runtime errors during the computation. The first type of error is due to accessing an undefined member of an object or invoking a object property that is not a function. We use a special configuration error to denote the result of the computation as shown in Figure 10. We will show that a well-typed program will not result in error. The second type of error is due to deferencing a null pointer, which is represented by a special configuration nullPtrEx as shown in Figure 11. We tolerate this type of error.

$$\frac{P = {Fn_i}^{i \in 1..n}\ s \quad \chi' = \{\mathsf{FT} \mapsto {Fn_i}^{i \in 1..n}\} \quad \emptyset, \chi', s \rightsquigarrow H, \chi}{\emptyset, \emptyset, P \rightsquigarrow H, \chi} \text{ R-Prog}$$

$$\frac{\begin{array}{c}\mathsf{lookup}(F, \chi(\mathsf{FT})) = \mathsf{function}\ F(x_p)\{s\} \\ \chi' = \{\mathtt{this} \mapsto \iota, x_p \mapsto \chi(y), \mathsf{FT} \mapsto \chi(\mathsf{FT})\} \\ \iota \notin \mathsf{dom}(H) \quad H[\iota \mapsto \{\ \}], \chi', s \rightsquigarrow H', \chi''\end{array}}{H, \chi, x = \mathsf{new}\ F(y) \rightsquigarrow H', \chi[x \mapsto \iota]} \text{ R-New}$$

$$\frac{H(\chi(y))(m) = v}{H, \chi, x = y.m \rightsquigarrow H, \chi[x \mapsto v]} \text{ R-Sel}$$

$$\frac{\begin{array}{c}H(\chi(y))(m) = f \\ \mathsf{lookup}(f, \chi(\mathsf{FT})) = \mathsf{function}\ f(x_p)\{s; \mathsf{return}\ y''; \} \\ \chi' = \{\mathtt{this} \mapsto \chi(y), x_p \mapsto \chi(y'), \mathsf{FT} \mapsto \chi(\mathsf{FT})\} \\ H, \chi', s \rightsquigarrow H', \chi''\end{array}}{H, \chi, x = y.m(y') \rightsquigarrow H', \chi[x \mapsto \chi''(y'')]} \text{ R-Invk}$$

$$\frac{H(\chi(y)) = o \quad H' = H(\chi(y) \mapsto o[m \mapsto \chi(y')])}{H, \chi, y.m = y' \rightsquigarrow H', \chi} \text{ R-Upd}$$

$$H, \chi, x = y \rightsquigarrow H, \chi[x \mapsto \chi(y)] \text{ R-Asn}$$

$$\frac{z = n \mid b \mid f}{H, \chi, x = z \rightsquigarrow H, \chi[x \mapsto z]} \text{ R-Asn2}$$

$$\frac{H, \chi, s \rightsquigarrow H', \chi' \quad H', \chi', s' \rightsquigarrow H'', \chi''}{H, \chi, s; s' \rightsquigarrow H'', \chi''} \text{ R-Seq}$$

$$\frac{\begin{array}{c}H, \chi, s_1 \rightsquigarrow H_1, \chi_1 \quad H, \chi, s_2 \rightsquigarrow H_2, \chi_2 \\ (\chi(x) = \mathtt{true} \wedge i = 1) \vee (\chi(x) = \mathtt{false} \wedge i = 2) \\ \mathtt{trim}(\chi_1, \chi_2, \chi_1', \chi_2')\end{array}}{H, \chi, \mathtt{if}(x)\{s_1\}\ \mathtt{else}\{s_2\} \rightsquigarrow H_i, \chi_i'} \text{ R-If}$$

Figure 9. Operational semantics where the reduction rules of statements assume an implicit function table FT that maps each function/constructor name to its declaration.

A.2 Type soundness

For type soundness proof, we define an invariant that holds in each reduction step. The invariant is written as $\Sigma, \Gamma, C \vdash H, \chi$, which means that the heap H and stack χ are well-formed under the environments Σ and Γ, and constraint set C, where Σ maps object labels to their types: $\Sigma = \{\iota_i \mapsto {\varsigma_i}^{i \in 1..n}\}$.

The judgment $\Sigma, \Gamma, H, C \vdash v : \tau$ denotes that the value v is well-typed with the type τ. Here we use a function $\mathsf{Lower}_C(\tau)$ to retrieve the set of lower bound types of τ.

$$\mathsf{Lower}_C(t) = \bigcup_{\{\tau \le t \in \mathsf{cls}(C)\}} \mathsf{Lower}_C(\tau)$$

$$\frac{\tau \neq t \ \wedge\ \tau \neq (t, M)}{\mathsf{Lower}_C(\tau) = \{\tau\}} \qquad \mathsf{Lower}_C((t, M)) = \mathsf{Lower}_C(t)$$

Given a constraint set C, a value is well-typed with the type τ if the lower bounds of τ in C includes the actual type of the value.

$$\frac{\mathsf{num} \in \mathsf{Lower}_C(\tau)}{\Sigma, \Gamma, H, C \vdash n : \tau} \qquad \frac{\mathsf{bool} \in \mathsf{Lower}_C(\tau)}{\Sigma, \Gamma, H, C \vdash b : \tau} \qquad \frac{\mathsf{null} \in \mathsf{Lower}_C(\tau)}{\Sigma, \Gamma, H, C \vdash \mathsf{null} : \tau}$$

$$\frac{H(\chi(y))(m) = \mathsf{undef}}{H, \chi, x = y.m \rightsquigarrow \mathsf{error}}$$

$$\frac{H, \chi, s \rightsquigarrow \mathsf{error} \text{ or } (H, \chi, s \rightsquigarrow H', \chi' \ \wedge \ H', \chi', s' \rightsquigarrow \mathsf{error})}{H, \chi, s; s' \rightsquigarrow \mathsf{error}}$$

$$\frac{\begin{array}{c} H(\chi(y))(m) = \mathsf{undef} \text{ or} \\ H(\chi(y))(m) = f \ \wedge \ f \notin \mathsf{dom}(\chi) \text{ or} \\ \chi(f) = \mathsf{function}\ f(x_p)\{s; \mathsf{return}\ y''; \} \\ \chi' = \{\mathtt{this} \mapsto \chi(y), x_p \mapsto \chi(y')\} \quad H, \chi', s \rightsquigarrow \mathsf{error} \end{array}}{H, \chi, x = y.m_j(y') \rightsquigarrow \mathsf{error}}$$

$$\frac{\begin{array}{c} F \notin \mathsf{dom}(\chi) \text{ or} \\ \chi(F) = \mathsf{function}\ F(x_p)\{s\} \quad \iota \notin \mathsf{dom}(H) \quad H' = H[\iota \mapsto \{\ \}] \\ \chi' = \{\mathtt{this} \mapsto \iota, x_p \mapsto \chi(y)\} \quad H', \chi', s \rightsquigarrow \mathsf{error} \end{array}}{H, \chi, x = \mathsf{new}\ F(y) \rightsquigarrow \mathsf{error}}$$

Figure 10. Error of accessing undefined members or functions

$$\frac{\chi(y) = \mathsf{null}}{H, \chi, x = y.m \rightsquigarrow \mathsf{nullPtrEx}}$$

$$\frac{\chi(y) = \mathsf{null}}{H, \chi, y.m = y' \rightsquigarrow \mathsf{nullPtrEx}}$$

$$\frac{H, \chi, s \rightsquigarrow \mathsf{nullPtrEx} \text{ or } H, \chi, s \rightsquigarrow H', \chi' \quad H', \chi', s' \rightsquigarrow \mathsf{nullPtrEx}}{H, \chi, s; s' \rightsquigarrow \mathsf{nullPtrEx}}$$

$$\frac{\begin{array}{c} \chi(y) = \mathsf{null} \text{ or} \\ H(\chi(y))(m) = f \quad \chi(f) = \mathsf{function}\ f(x_p)\{s; \mathsf{return}\ y''; \} \\ \chi' = \{\mathtt{this} \mapsto \chi(y), x_p \mapsto \chi(y')\} \quad H, \chi', s \rightsquigarrow \mathsf{nullPtrEx} \end{array}}{H, \chi, x = y.m_j(y') \rightsquigarrow \mathsf{nullPtrEx}}$$

$$\frac{\begin{array}{c} \iota \notin \mathsf{dom}(H) \quad H' = H[\iota \mapsto \{\ \}] \quad \chi(F) = \mathsf{function}\ F(x_p)\{s\} \\ \chi' = \{\mathtt{this} \mapsto \iota, x_p \mapsto \chi(y)\} \quad H', \chi', s \rightsquigarrow \mathsf{nullPtrEx} \end{array}}{H, \chi, x = \mathsf{new}\ F(y) \rightsquigarrow \mathsf{nullPtrEx}}$$

Figure 11. Null pointer exception

Given a constraint set C, a function value is well-typed with the type τ, if after renaming of variables, the constraint set of the function value is included in C and the renamed function type is a lower bound of τ in C.

$$\frac{\begin{array}{c} \Gamma(f) = \tau_f \setminus C_f \quad \Psi \text{ is a renaming of the variables in } C_f \\ \Psi\tau_f \in \mathsf{Lower}_C(\tau) \quad \Psi C_f \subseteq C \end{array}}{\Sigma, \Gamma, H, C \vdash f : \tau}$$

Recall that a constraint $m \in \tau$ represents the reading of m on τ and this access is safe given a constraint set C if $C \cup \{m \in \tau\}$ is consistent. We use the judgement $C \vdash m \in \tau$ to represent this.

$$\frac{C \cup \{m \in \tau\} \text{ is consistent}}{C \vdash m \in \tau}$$

The environment Σ is only used in the proof of type soundness and we make sure that it maps each object label ι to the most current pro-type associated with ι. For a label ι to be well-typed with the type τ, $\Sigma(\iota)$ must be a lower bound of τ in C. Also, for each m that can be accessed through τ, $H(\iota)(m)$ must be well-typed for any type variable t with the constraint set $C \cup \{\tau \leq [m^+ : t]\}$. This constraint $\tau \leq [m^+ : t]$ represents a read access of m on τ.

$$\frac{\begin{array}{c} \Sigma(\iota) = \varsigma \quad \varsigma \in \mathsf{Lower}_C(\tau) \\ \forall m.\ C \vdash m \in \tau \Rightarrow \Sigma, \Gamma, H, C \cup \{\tau \leq [m^+ : t]\} \vdash H(\iota)(m) : t \end{array}}{\Sigma, \Gamma, H, C \vdash \iota : \tau}$$

We define a well-formed type τ with the judgement $C \vdash \tau$. An extended type (τ, M) is well-formed if for each member m in M, there exists a corresponding constraint $\tau \leq [m^- : \_]$ in the closure of the current constraint set C. This constraint represents the writing of the member m to τ. Other kinds of types are well-formed by default.

$$\frac{\forall m \in \mathsf{Upper}_C(M).\ (\tau, M) \leq [m^- : \_] \in \mathsf{cls}(C)}{C \vdash (\tau, M)} \qquad \frac{\tau \neq (t, M)}{C \vdash \tau}$$

Using the above definitions, we define the program invariant as:

$$\frac{\begin{array}{c} \forall \iota.\ \iota \in \mathsf{dom}(\Sigma) \Leftrightarrow \iota \in \mathsf{dom}(H) \\ \forall y.\ y \in \mathsf{dom}(\Gamma) \Leftrightarrow y \in \mathsf{dom}(\chi) \\ \forall Fn \in \chi(\mathsf{FT}).\ \Gamma_{\mathtt{init}} \vdash Fn \\ \forall y \in \mathsf{dom}(\chi).\ \Gamma(\Gamma(y)) = \tau \quad \Sigma, \Gamma, H, C \vdash \chi(y) : \tau \quad C \vdash \tau \end{array}}{\Sigma, \Gamma, C \vdash H, \chi}$$

The judgment $\Sigma, \Gamma, C \vdash H, \chi$ says that the heap H and stack χ are well-formed with respect to the environments Σ and Γ, and constraint set C. For this invariant to hold, the domains of H and Σ must be the same and the domains of χ and Γ have the same set of variables. Also, each variable in χ must be well-typed and each definition in $\chi(\mathsf{FT})$ is well-typed with the initial environment $\Gamma_{\mathtt{init}}$, which only contains type mappings for functions and constructors.

Lemma A.1 shows that the execution of a well-typed statement cannot lead to errors caused by accessing undefined object members or functions. Also, the execution of a well-typed statement will result in a well-formed heap and stack.

Lemma A.1. *If* $\Sigma, \Gamma, C \vdash H, \chi$ *and* $\Gamma, C \vdash s \ \| \ \Gamma', C'$*, then* $H, \chi, s \not\rightsquigarrow \mathsf{error}$*, and if* $H, \chi, s \rightsquigarrow H', \chi'$*, then* $\exists\ \Sigma'$ *such that* $\Sigma', \Gamma', C' \vdash H', \chi'$.

The proof is by straightforward induction and is omitted.

From Lemma A.1, we can conclude that well-typed programs will not lead to errors caused by accessing undefined members and invoking object properties that are not functions.

Theorem A.2 (Type Soundness)**.** *If* $\Gamma \vdash P \ \| \ \Gamma', C$*, then* $\emptyset, \emptyset, P \not\rightsquigarrow \mathsf{error}$ *and if* $\emptyset, \emptyset, P \rightsquigarrow H, \chi$*, then* $\exists\ \Sigma$ *such that* $\Sigma, \Gamma', C \vdash H, \chi$.