Type Inference for Scripting language with Implicit Extension

Tian Zhao
University of Wisconsin – Milwaukee
tzhao@uwm.edu

***Abstract*—This paper presents a constraint-based type inference algorithm for a subset of the JavaScript language. 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. We prove that a program is typable if and only if we can infer its types. We also extend the type system to allow strong updates and unrestricted extensions to new objects.**

I. Introduction

JavaScript is a widely used scripting languages for Web applications. As a dynamic language, it has some flexible language features such as method update and method/field additions. These features are also potential sources of runtime errors such as accessing undefined members of objects. Since JavaScript does not have static types, there is no way to determine statically which members have been added to an object at each program point and programmers have to rely on documentation or other tools to avoid these types of mistakes.

Past research have proposed the use of static types to keep track of object extensions and there are some design variations. One approach, taken by Anderson et al.'s type inference algorithm [1], is to use flow-sensitive object types that distinguish two types of object members: definite members (ones that have been defined) and potential members (ones that may be defined later). Only definite members may be accessed while potential members may become definite after object extensions. This design allows objects be extended at any time. Another approach, seen in Recency Types [2] and Bono and Fisher's calculus with object extensions [3], is to use two sets of object types: one set allows object extension while the other set does not. The idea is to model objects at initialization stage using extensible object types and after that, the objects are given fixed types. With this design, the extensions made to objects at initialization stage are not restricted by the initial types of the objects. To support this behavior and also allow objects be extended after initialization stage, it is possible to have features of both approaches in one type system [4].

In this paper, we present a type system and a type inference algorithm based on the last design choice to have two sets of object types. One set consists of singleton types assigned to new objects in local scope to allow strong updates where members of an object can be replaced by values of different types and to permit unrestricted extensions. The other set consists of flow-sensitive obj-types that distinguish definite and potential members. We allow a variable of a singleton type and a variable of an obj-type to point to the same object though the two types must be compatible so that their common members cannot have strong updates. However, the variable of the singleton type can still have strong update on other members and have unrestricted extensions.

Our type system keeps track of members added to an object by both explicit add operation, and implicit extension through function parameters and self-inflicted extension [4], which is the extension that an object made to itself upon receiving a message as shown in the following example.

A. Motivating examples

```
1  function Form(a) {
2     this.id = a;
3     this.set = setter;
4  }
5  function setter(b) {
6     this.handle = b;
7     return 0;
8  }
9  function handler(c) {
10    // do something
11    return 0;
12 }
13 // main
14 x = new Form(1);
15 z = x.handle(1);            // error
16 y = x.set(handler);
17 z = x.handle(1);            // OK
```

Fig. 1. Example of self-inflected extension

Figure 1 is an example of self-inflicted extension, where `Form` is a constructor function that return new objects with a field `id` and a method `set`, which adds a handle method to the current object. In the main program, when `handle` is called at line 15, there should be a runtime error since `handle` is not yet defined in the `Form` object `x`. However, it is OK to call `handle` for the 2nd time (line 17) since `set` has added this method to `x`. Anderson's algorithm [1] does not allow this call since it only considers members added to objects by explicit add operations while `handle` is added indirectly by the method `set`. Our type system keeps track of both types of object extensions. Consequently, it can determine that the variable `x` at line 17 refers to an object with the method `handle`.

We also consider an extension to our type system to support strong updates to new objects where an object's member may be replaced by a value of a different type. Since obj-types are not extensible, object extensions are limited by the potential members in the object types. Also, strong updates to definite members are not allowed. This is not a problem for an empty object since we can give it a type with any potential members. However, new objects instantiated from a constructor function have the same type – the return type of the constructor function, so that potential extensions made to these objects are limited by this type – the types of the definite members cannot be changed. JavaScript allows constructor functions to return any objects though, in many cases, the expected behavior of a constructor function is to return a new object each time it is called through the `new` operator. For other cases, one can make an ordinary function call instead. Therefore, we only consider this behavior of the constructor functions. We extend our type system with a kind of singleton types to support strong updates and unrestricted extensions to new objects.

```
1 function F(x) {
2     this.a = 1;
3     this.b = "one";
4 }
5 x1 = new F(0);
6 x2 = new F(0);
7 x1.b = true;
8 x1.c = 2;
9 x2.c = false;
```

Fig. 2. Strong updates and unrestricted extensions

For example, the program in Figure 2 creates two `F` objects `x1` and `x2`, and extends `x1` and `x2` with field `c` of integer type and boolean type respectively. Also, the member `b` of object `x1` received a strong update – its type is changed from string to boolean. We can allow this by assigning singleton types to `x1` and `x2`.

In summary, we make the following contributions:

- a sound and complete type inference algorithm for a small subset of JavaScript language to keep track of new members added to objects through add operation and implicit extensions.
- an extension to our inference algorithm to allow strong updates and unrestricted extensions to new objects

In the rest of paper, we first give an informal discussion of our approach in Section II. Next, we formalize a type system on a small subset of JavaScript to support self-inflicted extension. We present the syntax and type rules in Section III. We explain the details of type inference algorithm and its correctness in Section IV. We explain the extension to add singleton types to new objects in Section V. The operational semantics, inference rules, and a type inference example are in the appendix.

II. Approach

We follow the design of Anderson's type system [1] by labeling each member of an object type as potential or definite to indicate whether the member is possibly defined or definitely exists respectively. The labels are inferred along with the types of a program.

Consider the example in Figure 1, the type of the variable `x` at line 14 is

$$t_{\mathtt{x}} = [\mathsf{id} : (\mathsf{int}, \bullet), \mathtt{set} : (t_{\mathtt{setter}}, \bullet), \mathtt{handle} : (t_{\mathtt{handler}}, \circ)]$$

where $\bullet$ labels definite members while $\circ$ labels potential members. Each distinct object type and function type has a name and is defined with an equation, where the right-hand-side shows the structure of the type. Each type name may be referenced in the definitions of some other types.

We support width subtyping of object types but not depth subtyping. For example, type $t_{\mathtt{x}}$ is a subtype of t where $t = [\mathtt{set} : (t_{\mathtt{setter}}, \bullet)]$. Also, it is safe to give an object with a member `m` a type that labels `m` as potential. For example, t is a subtype of t' where $t' = [\mathtt{set} : (t_{\mathtt{setter}}, \circ)]$.

Notice that the `handle` method of $t_{\mathtt{x}}$ is a potential method only and it is illegal to call methods with such a label (e.g. line 15 of Figure 1). A potential member becomes definite after an assignment. The function call at Line 16 adds the `handle` method to `x`. The type of `x` at line 17 is

$$t'_{\mathtt{x}} = [\mathsf{id} : (\mathsf{int}, \bullet), \mathtt{set} : (t_{\mathtt{setter}}, \bullet), \mathtt{handle} : (t_{\mathtt{handler}}, \bullet)].$$

At this point, it is then safe to call `handle`. In our type system, we represent $t'_{\mathtt{x}}$ as an *extended type* – $(t_{\mathtt{x}}, \{\mathtt{handle}\})$, where the set $\{\mathtt{handle}\}$ includes the member `handle` added to the object. This representation makes it easier to keep track of members added to `this` pointer and function parameters.

The method call `x.set(handler)` updates the receiver object `x` with the function `handler`. To obtain the information about which members are added to the receiver object, we define function types in the form of

$$\tau \times \tau' \to u$$

where τ is the type of self pointer, τ' and u are parameter and return type respectively. The meta variable u ranges over object types, function types, and primitive types while $\tau = (u, M)$ ranges over extended types and M is ignored if u is not an object type. In an extended type $\tau = (t, M)$ of self or function parameter, M is the set of members added to self or the function parameter during the function call. The type of `setter` is then

$$t_{\mathtt{setter}} = (t, \{\mathtt{handle}\}) \times (t_{\mathtt{handler}}, \emptyset) \to \mathsf{int}$$

where t and $t_{\mathtt{handle}}$ are defined as

$$t = [\mathtt{handle} : (t_{\mathtt{handler}}, \circ)]$$

$$t_{\mathtt{handler}} = (\mathsf{top}, \emptyset) \times (\mathsf{int}, \emptyset) \to \mathsf{int}$$

where top is the super type of all object types.

To allow strong updates and unrestricted extensions as in Figure 2, we define a form of singleton types ς, where we

label object members that can receive strong update with $*$. As shown below, ς_{F} is the return type of the constructor `F` and $\varsigma_{\mathtt{x1}}$ and $\varsigma_{\mathtt{x2}}$ are the types of `x1` and `x2` after the last assignment.

$$\varsigma_{\mathsf{F}} = @[\mathsf{a} : (\mathsf{int}, *), \mathsf{b} : (\mathsf{string}, *)]$$
$$\varsigma_{\mathtt{x1}} = @[\mathsf{a} : (\mathsf{int}, *), \mathsf{b} : (\mathsf{bool}, *), \mathsf{c} : (\mathsf{int}, *)]$$
$$\varsigma_{\mathtt{x2}} = @[\mathsf{a} : (\mathsf{int}, *), \mathsf{b} : (\mathsf{string}, *), \mathsf{c} : (\mathsf{bool}, *)]$$

The singleton types are only assigned to new objects and local variables that reference these objects. For simplicity, the types of object members, function parameters, and function return types (other than the constructor function's return type) are obj-types. We keep track of the aliases of singleton types within local scope and they receive obj-types once they are assigned to some objects' fields or passed as parameters to other functions. The singleton type of an object has to be updated once the object is assigned to some variable of obj-types. Consider the following example where the variable `x` is passed to function `f`.

```
1 function f(y) {
2    y.a = 1;
3    return y;
4 }
5 x = new F(0);
6 z = f(x);
7 x.b = true;
8 x.c = 2;
```

If the type of `x` starts with $\varsigma_{\mathtt{x}}$, then it becomes $\varsigma'_{\mathtt{x}}$ after the call.

$$\varsigma_{\mathtt{x}} = @[\mathsf{a} : (\mathsf{int}, *), \mathsf{b} : (\mathsf{string}, *)],$$
$$\varsigma'_{\mathtt{x}} = @[\mathsf{a} : (\mathsf{int}, \bullet), \mathsf{b} : (\mathsf{string}, *)]$$

In effect, the type system has to change the label of `x.a` so that it can no longer receive strong updates. Still, variable `x` can have strong updates on its member `b` and be extended with additional members so that its type eventually becomes

$$\varsigma''_{\mathtt{x}} = @[\mathsf{a} : (\mathsf{int}, \bullet), \mathsf{b} : (\mathsf{bool}, *), \mathsf{c} : (\mathsf{int}, *)].$$

A singleton type may also have potential members as well. Consider the example below where the variable `y` may point to the object `x1` or `x2` and `y` is assigned a new member `c`.

```
1  x1 = new F(0);
2  x2 = new F(0);
3  x1.b = true;
4  if (z > 0) y = x1; else y = x2;
5  y.c = 1;
```

The variable `y` is given different singleton types in the branches and at line 5, we have to merge the two singleton types into an obj-type $t_{\mathtt{y}} = [\mathsf{c} : (\mathsf{int}, \circ)]$ since a singleton type cannot be assigned to more than one object. The types of `x1` and `x2` after the `if` statement become

$$\varsigma_{\mathtt{x1}} = [\mathsf{a} : (\mathsf{int}, *), \mathsf{b} : (\mathsf{bool}, *), \mathsf{c} : (\mathsf{int}, \circ)]$$
$$\varsigma_{\mathtt{x2}} = [\mathsf{a} : (\mathsf{int}, *), \mathsf{b} : (\mathsf{string}, *), \mathsf{c} : (\mathsf{int}, \circ)]$$

since both of them must be subtype of `y`, which has the potential member `c`. After the `if` statement, the variables `x1` and `x2` can still be extended with new members and their `a` and `b` members can still receive strong updates.

Finally, the following example illustrates the interaction between singleton type and implicit extensions.

```
1 function G(i) {
2   this.a = i;
3   this.m = g;
4 }
5 function g(j) {
6   this.b = j;
7   return this;
8 }
9 x = new G(0);
10 y = x.m(1);
```

The variable `x`'s type on line 9 is

$$\varsigma_{\mathtt{x}} = @[\mathsf{a} : (\mathsf{int}, *), \mathtt{m} : (t_{\mathtt{g}}, *)],$$

where $t_{\mathtt{g}} = (t, \{b\}) \times (\mathsf{int}, \emptyset) \rightarrow t'$, $t = [\mathsf{b} : (\mathsf{int}, \circ)]$, and $t' = [b : (\mathsf{int}, \bullet)]$. After line 10, the type of `x` becomes

$$\varsigma'_{\mathtt{x}} = @[\mathsf{a} : (\mathsf{int}, *), \mathtt{m} : (t_{\mathtt{g}}, *), \mathsf{b} : (\mathsf{int}, \bullet)]$$

The variable `x` is extended with a member `b` through implicit extension. In fact, `y` is an alias of `x` but `y` has an obj-type since we do not track singleton type across function calls.

III. Formalization

In this section, we present a formalization of our type system. The details of type inference are covered in Section IV. This formalization is for implicit extension only. Additions to the type system are discussed in Section V.

A. 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 model 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 [5].

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. 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. For simplicity, we assume that function parameter (denoted by x_p) is read-only.

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

Fig. 3. Syntax

B. Static semantics

The meta variable u ranges over four kinds of types: function type, object type, primitive type (such as int and bool), and a top type.

u	::=	$t \mid \mathsf{int} \mid \mathsf{bool} \mid \mathsf{top}$	
τ	::=	(u, M)	extended types

The variable t ranges over the names of function and object types, which are defined by equations of the form:

t	=	$[m_i : (u_i, \psi_i)^{i\in i..n}]$	object type
t	=	$\tau \times \tau' \to u$	function type
ψ	::=	$\circ$	potential
	|	$\bullet$	definite

The meta variable ψ ranges over the label $\circ$ and $\bullet$, which indicates whether a member is potentially or definitely present.

a) Subtyping: The subtyping relation is reflexive and transitive.

$$u \le u \qquad \frac{u \le u' \quad u' \le u''}{u \le u''} \qquad \frac{\tau \le u \quad u \le u'}{\tau \le u'}$$

All object and function types are subtype of top. An extended type (u, M) is a subtype of u if u is not an object type.

$$t \le \mathsf{top} \quad (\mathsf{int}, \_) \le \mathsf{int} \quad (\mathsf{bool}, \_) \le \mathsf{bool} \quad (t_f, \_) \le t_f$$

where t_f represents function types.

A function type is a subtype of another one if they are structurally equivalent. For simplicity, we do not have covariant return type and contravariant parameter type for function types. Let ft range over $\tau \times \tau' \to u$.

$$\frac{t = \mathtt{ft} \quad t' = \mathtt{ft}}{t \le t'} \qquad \frac{t = \mathtt{ft}}{t \le \mathtt{ft}} \qquad \frac{t = \mathtt{ft}}{(t, \_) \le \mathtt{ft}}$$

The expression $t(m)$ returns the type information of member m in object type t if it is defined in t, otherwise, $t(m)$ is undefined.

$$\frac{t = [\ldots m : (u, \psi) \ldots]}{t(m) = (u, \psi)} \quad t(m) = \mathsf{undef}\ \text{otherwise, where}$$

undef is used here to denote undefined member of a type.

We define a subtyping relation between extended object type (t, M) and object type t' as below, where member labels follow the partial order $\psi \le \psi$ and $\bullet \le \circ$.

$$\frac{\forall m.\ t'(m) = (u, \psi') \Rightarrow (t(m) = (u, \psi)\ \wedge\ (\psi \le \psi'\ \vee\ m \in M))}{(t, M) \le t'}$$

We also define a subtyping relation below for the convenience of stating typing rules.

$$\frac{t(m) = (u, \psi) \quad (\psi \le \psi'\ \vee\ m \in M)}{(t, M) \le [m : (u, \psi')]}$$

1) Type rules for functions and constructors: We use the symbol Γ to represent type environment that maps variables, function/constructor names, and constants to their types. For any variable or name in the domain of Γ, we define

$$\frac{\Gamma = [\ldots z \mapsto \tau \ldots]}{\Gamma(z) = \tau}$$

If the function f has type t, we let $\Gamma(f) = (t, \emptyset)$. Also, $\Gamma(n) = (\mathsf{int}, \emptyset)$ and $\Gamma(b) = (\mathsf{bool}, \emptyset)$ for any n and b.

A judgment of the form $\Gamma \vdash s \parallel \Gamma'$ says that the statement s is well-typed with the environment Γ and the execution of s will result in a (possibly new) environment Γ'.

Figure 4 shows the typing rule for program and functions, where a program P with environment Γ is well-typed if its functions, constructors, and main statement are well-typed with Γ. The environment for typing a program includes the mapping of functions and constructors to their types.

A function f is well-typed given an environment Γ if we can construct a new environment for the function body so that it is well-typed. In particular, Γ maps `this` variable (or parameter) to an extended type with empty member set such as $(u, \emptyset)$. After the function body s is executed, the new environment may map `this` (or parameter) to another extended type such as (u, M). We use (u, M) as the type of `this` (or parameter) in the function type. M is used to record the members added to `this` (or parameter) in s.

For a constructor to be well-typed, the type of `this` pointer before the execution of the constructor body must not have any definite members since the constructor is always invoked with an empty receiver object. Rule (T-Ctr) uses a function $\mathsf{def}(u)$ to return the set of definite members of u, where $\mathsf{def}(t) = \{m \mid t(m) = (\_, \bullet)\}$ and $\mathsf{def}(\mathsf{top}) = \emptyset$.

2) Type rules for statements: Figure 5 lists the type rules for statements.

Rule (T-Dec) says that each variable declaration defines a new variable not already in the domain of the type environment, where dom is a function that returns the domain of a

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

$$\frac{\begin{array}{c}\Gamma[\texttt{this} \mapsto (u_{\texttt{this}}, \emptyset), x_p \mapsto (u_{arg}, \emptyset)] \vdash s \ \parallel \ \Gamma' \\ \Gamma'(z) \leq u \quad \Gamma(f) \leq \Gamma'(\texttt{this}) \times \Gamma'(x_p) \to u\end{array}}{\Gamma \vdash \mathsf{function} \ f(x_p)\{s; \mathsf{return} \ z\}} \quad \text{T-Fn}$$

$$\frac{\begin{array}{c}\Gamma[\texttt{this} \mapsto (u_{\texttt{this}}, \emptyset), x_p \mapsto (u_{arg}, \emptyset)] \vdash s \ \parallel \ \Gamma' \\ \mathsf{def}(u_{\texttt{this}}) = \emptyset \\ \Gamma'(\texttt{this}) \leq u \quad \Gamma(F) = \Gamma'(x_p) \to u\end{array}}{\Gamma \vdash \mathsf{function} \ F(x_p)\{s\}} \quad \text{T-Ctr}$$

Fig. 4. Typing rules for program, constructor, and function

$$\frac{x \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \mathsf{var} \ x \ \parallel \ \Gamma[x \mapsto (u, \emptyset)]} \quad \text{T-Dec}$$

$$\frac{\Gamma' = \Gamma[x \mapsto \Gamma(z)]}{\Gamma \vdash x = z \ \parallel \ \Gamma'} \quad \text{T-Assn}$$

$$\frac{\begin{array}{c}\Gamma(y) \leq [m_j : (u_j, \circ)] \quad \Gamma(z) \leq u_j \\ \Gamma(y) = (t, M) \quad M' \subseteq M \cup \{m_j\}\end{array}}{\Gamma \vdash y.m_j = z \ \parallel \ \Gamma[y \mapsto (t, M')]} \quad \text{T-Upd}$$

$$\frac{\Gamma(F) = \tau_{arg} \to u \quad \Gamma \vdash z : \tau_{arg} \ \parallel \ \Gamma'}{\Gamma \vdash x = \mathsf{new} \ F(z) \ \parallel \ \Gamma'[x \mapsto (u, \emptyset)]} \quad \text{T-New}$$

$$\frac{\Gamma(y) \leq [m_j : (u_j, \bullet)]}{\Gamma \vdash x = y.m_j \ \parallel \ \Gamma[x \mapsto (u_j, \emptyset)]} \quad \text{T-Sel}$$

$$\frac{\begin{array}{c}\Gamma(y) \leq [m_j : (t_j, \bullet)] \\ t_j \leq \tau_{\texttt{this}} \times \tau_{arg} \to u_{res} \\ \Gamma \vdash z : \tau_{arg} \ \parallel \ \Gamma' \quad \Gamma; \Gamma' \vdash y : \tau_{\texttt{this}} \ \parallel \ \Gamma''\end{array}}{\Gamma \vdash x = y.m_j(z) \ \parallel \ \Gamma''[x \mapsto (u_{res}, \emptyset)]} \quad \text{T-Invk}$$

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

$$\frac{\begin{array}{c}\Gamma \vdash s_1 \ \parallel \ \Gamma_1 \quad \Gamma \vdash s_2 \ \parallel \ \Gamma_2 \\ \Gamma(z) \leq \mathsf{bool} \quad \mathtt{merge}(\Gamma_1, \Gamma_2, \Gamma')\end{array}}{\Gamma \vdash \mathtt{if}(z) \ \{s_1\}\mathtt{else}\{s_2\} \ \parallel \ \Gamma'} \quad \text{T-If}$$

Fig. 5. Type rules for statements

mapping. Once a variable is declared, we assign a type to that variable in the environment though the type may be changed later by assignments. Rule (T-Upd) applies to the member update/add operation of the form $y.m_j = z$. We extend the type of y so that m_j can be read after the statement is executed (regardless of the original label of m_j). Rule (T-New) uses the return type of the constructor function to replace the type of the variable that the new object is assigned to.

Rule (T-Sel) requires the selected member to be definite. Rule (T-Invk) also requires the called method to be definite and the receiver object's type to be a subtype of `this` pointer of the called method. Note that both rules involve passing arguments to parameters that may extend the argument objects with new members. We define the judgment $\Gamma \vdash z : \tau \ \parallel \ \Gamma'$ to record the change to the type of a variable z after it is passed as an argument to a parameter of type τ where Γ' is the new environment after updating the type of z. The judgment $\Gamma; \Gamma' \vdash y : (u, M) \ \parallel \ \Gamma''$ is for similar purpose for the receiver object y except that $\Gamma'(y)$ may be different from $\Gamma(y)$ if y references the same object as the argument z does.

$$\frac{z = n \mid b \mid f \quad \Gamma(z) \leq u}{\Gamma \vdash z : (u, \_) \ \parallel \ \Gamma}$$

$$\frac{\Gamma(y) \leq u \quad \Gamma(y) = (u_y, M_y) \quad M'_y \subseteq M \cup M_y}{\Gamma \vdash y : (u, M) \ \parallel \ \Gamma[y \mapsto (u', M'_y)]}$$

$$\frac{\Gamma(y) \leq u \quad \Gamma'(y) = (u_y, M_y) \quad M'_y \subseteq M \cup M_y}{\Gamma; \Gamma' \vdash y : (u, M) \ \parallel \ \Gamma'[y \mapsto (u_y, M'_y)]}$$

For example, consider the following program:

```
1  function f(x) {
2     this.m1 = true;
3     x.m2 = 2;
4     return x;
5  }
6  function F() {
7     this.m = f;
8  }
9  z1 = new F();
10 z2 = new F();
11 z = z1.m(z2);
12 y = z.m2;
```

where f has the type $t_f = (t_{\texttt{this}}, \{\texttt{m1}\}) \times (t_{arg}, \{\texttt{m2}\}) \to t_{res}$,

$$\begin{array}{rcl} t_{\texttt{this}} & = & [\texttt{m1} : (\mathsf{bool}, \circ)] \\ t_{arg} & = & [\texttt{m2} : (\mathsf{int}, \circ)] \\ t_{res} & = & [\texttt{m2} : (\mathsf{int}, \bullet)] \ . \end{array}$$

The type of `z1` and `z2` at line 10 is $(t_F, \emptyset)$,

$$t_F = [\texttt{m} : (t_f, \bullet), \texttt{m1} : (\mathsf{bool}, \circ), \texttt{m2} : (\mathsf{int}, \circ)] \ .$$

At line 11, we apply Rule (T-Invk) with the judgment $\Gamma \vdash \texttt{z2} : (t_{arg}, \{\texttt{m2}\}) \ \parallel \ \Gamma'$ to change the environment Γ to Γ', where $\Gamma = [\texttt{z1} : (t_F, \emptyset), \texttt{z2} : (t_F, \emptyset)]$ and $\Gamma' = [\texttt{z1} : (t_F, \emptyset), \texttt{z2} : (t_F, \{\texttt{m2}\})]$. Also, $\Gamma; \Gamma' \vdash \texttt{z1} : (t_{\texttt{this}}, \{\texttt{m2}\}) \ \parallel \ \Gamma''$ produces the environment $[\texttt{z1} : (t_F, \{\texttt{m1}\}), \texttt{z2} : (t_F, \{\texttt{m2}\})]$. After line 11, variable `z1` has a definite member `m1` and `z2` has a definite member `m2`.

Lastly, Rule (T-If) merges the type environments Γ_1 and Γ_2 generated by the branches s_1 and s_2 respectively into a new environment Γ' using the predicate `merge` defined below.

$$\mathtt{merge}(\Gamma, \Gamma, \Gamma) \quad \frac{\mathtt{merge}(\Gamma_1, \Gamma_2, \Gamma)}{\mathtt{merge}(\Gamma_2, \Gamma_1, \Gamma)}$$

$$\frac{y \notin \mathsf{dom}(\Gamma_2) \quad \mathtt{merge}(\Gamma_1, \Gamma_2, \Gamma)}{\mathtt{merge}((y \mapsto \tau, \Gamma_1), \Gamma_2, \Gamma)}$$

$$\frac{\begin{array}{c}\tau_1 = (u, M_1) \quad \tau_2 = (u, M_2) \\ \mathtt{merge}(\Gamma_1, \Gamma_2, \Gamma) \quad M \subseteq M_1 \cap M_2\end{array}}{\mathtt{merge}((y \mapsto \tau_1, \Gamma_1), (y \mapsto \tau_2, \Gamma_2), (y \mapsto (u, M), \Gamma))}$$

$$\frac{\begin{array}{c}\forall i = 1, 2. \ \tau_i \leq u \quad \tau_i = (u_i, M_i) \\ u_1 \neq u_2 \quad \mathtt{merge}(\Gamma_1, \Gamma_2, \Gamma))\end{array}}{\mathtt{merge}((y \mapsto \tau_1, \Gamma_1), (y \mapsto \tau_2, \Gamma_2), (y \mapsto (u, \emptyset), \Gamma))}$$

A variable of the type (u, M_1) and (u, M_2) in each branch maps to $(u, M_1 \cap M_2)$ in the merged environment. This allows us to keep track of the members added to self and parameter. For example, y in the example below has the type $(t_F, \{\texttt{m1}\})$ after line 7.

```
1 y = new F();
2 if (z) {
3   y.m1 = true;
4   y.m2 = 2;
5 } else {
6   y.m1 = false;
7 }
```

If a variable is mapped to (u_1, M_1) and (u_2, M_2) and $u_1 \neq u_2$, then it is mapped to $(u, \emptyset)$ where $(u_1, M_1) \leq u$ and $(u_2, M_2) \leq u$. This is illustrated in the following example, where the type of y in the merged environment is $(t_y, \emptyset)$ and $t_y = [\texttt{m1} : (\mathsf{bool}, \bullet)]$.

```
1 if (z) {
2   y = new F();
3   y.m1 = true;
4   y.m2 = 1;
5 } else {
6   y = new F();
7   y.m1 = false;
8 }
9 x = y.m1;
```

IV. Type Inference

The type inference algorithm includes three steps:

1) generate type constraints from a program,
2) apply closure rules to the constraint set until it is closed under the rules,
3) solve the closed constraint set.

The type inference rules are given in Figure 13 (see appendix D), where V ranges over type variables and $\mathcal{M}$ ranges over variables that correspond to sets of object members. The inference rules have similar structure as the typing rules. The first three rules in Figure 13 infer constraints from programs, functions, and constructors. The judgment $E \vdash_{\mathsf{inf}} P \mid \mathcal{C}$ infers a set of constraints $\mathcal{C}$ from a program P based on the initial environment E, where E maps function and constructor names to distinct type variables. Likewise, $E \vdash_{\mathsf{inf}} Fn \mid \mathcal{C}$ generates a set of constraints $\mathcal{C}$ from a function or constructor declaration Fn based on the environment E. Note that for each constructor function F, we create a a distinct type variable V_F for the initial type of `this` pointer. The inference rules make sure that each variable in the generated constraint set is unique.

The rest of the rules in Figure 13 generate type constraints from statements. Each judgment of the inference rule for statements is in the form of:

$$E \vdash_{\mathsf{inf}} s \;\parallel\; E' \mid \mathcal{C}$$

which generates a set of constraints $\mathcal{C}$ from a statement s with initial environment E and produces another environment E'. All environment E maps constants to $(V_{\mathsf{int}}, \mathcal{M}_\emptyset)$ or $(V_{\mathsf{bool}}, \mathcal{M}_\emptyset)$, constructor names to types of the form $(V, \mathcal{M}) \to V'$, function names to types of the form $(V, \mathcal{M}_\emptyset)$, and local variables to types of the form $(V, \mathcal{M})$. We add the constraints $\mathcal{M}_\emptyset \subseteq \emptyset$, $V_{\mathsf{int}} \leq \mathsf{int}$, and $V_{\mathsf{bool}} \leq \mathsf{bool}$ to the constraint set of a program.

The result of applying inference rules is a constraint set of the forms:

$$U \leq W \;\; \mathcal{M} \subseteq \texttt{mSet} \;\; \mathcal{M} = \mathcal{M}' \;\; \mathcal{M} \subseteq \mathcal{M} \cup \mathcal{M}' \;\; \mathcal{M} \subseteq \mathcal{M} \cap \mathcal{M}'$$

where the meta variable U and W are defined as

$$\begin{array}{rcl} U & ::= & V \mid (V, \mathcal{M}) \\ W & ::= & \texttt{bt} \mid V \mid [m : (V, \psi)] \mid (V_0, \mathcal{M}_0) \times (V_1, \mathcal{M}_1) \to V_2, \end{array}$$

`mSet` represents a set of member names such as $\emptyset$ and $\{m_i, m_j\}$, and `bt` represents base types int or bool.

A. Closure rules

To solve a constraint set $\mathcal{C}$, we first compute its closure.

Definition A constraint set $\mathcal{C}$ is closed if $\mathcal{C} \longrightarrow_A \mathcal{C}$.
Let $\mathsf{Closure}(\mathcal{C}) = \mathcal{C}'$ if $\mathcal{C} \longrightarrow_A^* \mathcal{C}'$ and $\mathcal{C}' \longrightarrow_A \mathcal{C}'$, where $\longrightarrow_A$ is defined by Rule 14 and $\longrightarrow_A^*$ is a transitive closure of $\longrightarrow_A$.

The closure relation $\longrightarrow_A$ depends on the following rules. Rule 1 applies transitive closure to subtyping relations.

$$U \leq V, V \leq W \longrightarrow U \leq W \tag{1}$$

Rule 2 to 4 ensure that a base type can only be subtype of itself.

$$V' \leq \texttt{bt}, (V', \mathcal{M}) \leq V \longrightarrow V \leq \texttt{bt} \tag{2}$$

$$V' \leq \texttt{bt}, V' \leq V \longrightarrow V \leq \texttt{bt} \tag{3}$$

$$(V, \mathcal{M}) \leq \texttt{bt} \longrightarrow V \leq \texttt{bt} \tag{4}$$

Rule 5 simplifies constraints for function types.

$$(V, \mathcal{M}) \leq \texttt{ft} \longrightarrow V \leq \texttt{ft} \tag{5}$$

where `ft` represents $(V_0, \mathcal{M}_0) \times (V_1, \mathcal{M}_1) \to V_2$.

Rule 6 and 7 check constraints on object and function types with common lower bounds.

$$\begin{array}{l} V \leq [m : (V', \_)], \\ V \leq [m : (V'', \_)] \end{array} \longrightarrow V' \leq V'', V'' \leq V' \tag{6}$$

$$\begin{array}{l} V \leq (V_0, \mathcal{M}_0) \times (V_1, \mathcal{M}_1) \to V_2 \\ V \leq (V_0', \mathcal{M}_0') \times (V_1', \mathcal{M}_1') \to V_2' \end{array} \longrightarrow \begin{array}{l} \forall i = 0, 1, 2. \\ V_i \leq V_i', V_i' \leq V_i \\ \forall j = 0, 1.\ \mathcal{M}_j = \mathcal{M}_j' \end{array} \tag{7}$$

Rule 8 to 11 propagate set membership for $\mathcal{M}$ variables.

$$\mathcal{M} = \mathcal{M}' \longrightarrow \mathcal{M}' = \mathcal{M} \tag{8}$$

$$\begin{array}{l} \mathcal{M}_1 \subseteq \texttt{mSet}_1, \mathcal{M}_2 \subseteq \texttt{mSet}_2, \\ \mathcal{M} \subseteq \mathcal{M}_1 \cup \mathcal{M}_2 \end{array} \longrightarrow \mathcal{M} \subseteq \texttt{mSet}_1 \cup \texttt{mSet}_2 \tag{9}$$

$$\mathcal{M}_i \subseteq \texttt{mSet}_i, \mathcal{M} \subseteq \mathcal{M}_1 \cap \mathcal{M}_2 \longrightarrow \mathcal{M} \subseteq \texttt{mSet}_i \tag{10}$$

$$\mathcal{M}' \subseteq \texttt{mSet},\ \mathcal{M} = \mathcal{M}' \longrightarrow \mathcal{M} \subseteq \texttt{mSet} \tag{11}$$

Rule 12 and 13 apply closure rules to member extension constraints.

$$(V, \mathcal{M}) \leq [m : (V', \psi)] \longrightarrow V \leq [m : (V', \circ)] \quad (12)$$

$$\frac{\mathcal{M} \subseteq \texttt{mSet} \quad m \notin \texttt{mSet}}{(V, \mathcal{M}) \leq [m : (V', \bullet)]} \longrightarrow V \leq [m : (V', \bullet)] \quad (13)$$

Rule 14 applies closure rules 1–13 to a constraint set to obtain a possibly larger constraint set.

$$\frac{\forall i \in 1..k.\ c_i \in \mathcal{C} \quad c_1, .., c_k \longrightarrow c'_1, .., c'_n}{\mathcal{C} \longrightarrow_A \mathcal{C} \cup \{c'_1, .., c'_n\}} \quad (14)$$

B. Constraint satisfiability

A solution S to a constraint set $\mathcal{C}$ maps each V in $\mathcal{C}$ to $\texttt{bt}$, t, or $\texttt{top}$, maps each $\mathcal{M}$ in $\mathcal{C}$ to set of member names, and

$$S((V_0, \mathcal{M}_0) \times (V_1, \mathcal{M}_1) \to V_2) = (S(V_0), S(\mathcal{M})) \times (S(V_1), S(\mathcal{M}_1)) \to S(V_2)$$

$$S(\texttt{bt}) = \texttt{bt} \quad S(V, \mathcal{M}) = (S(V), S(\mathcal{M}))$$

$$S([m : (V, \psi)]) = [m : (S(V), \psi)]$$

We say that a constraint set $\mathcal{C}$ is satisfiable if there exists a solution S to $\mathcal{C}$ such that

$$\begin{aligned}
U \leq W \in \mathcal{C} &\Rightarrow S(U) \leq S(W) \\
\mathcal{M} \subseteq \texttt{mSet} \in \mathcal{C} &\Rightarrow S(\mathcal{M}) \subseteq \texttt{mSet} \\
\mathcal{M} = \mathcal{M}' \in \mathcal{C} &\Rightarrow S(\mathcal{M}) = S(\mathcal{M}') \\
\mathcal{M} \subseteq \mathcal{M}' \cup \mathcal{M}'' \in \mathcal{C} &\Rightarrow S(\mathcal{M}) \subseteq S(\mathcal{M}') \cup S(\mathcal{M}'') \\
\mathcal{M} \subseteq \mathcal{M}' \cap \mathcal{M}'' \in \mathcal{C} &\Rightarrow S(\mathcal{M}) \subseteq S(\mathcal{M}') \cap S(\mathcal{M}'') \\
V_F \text{ appears in } \mathcal{C} &\Rightarrow \mathsf{def}(S(V_F)) = \emptyset
\end{aligned}$$

C. Constraint consistency

Before we solve a constraint set, we need to make sure it is consistent. We will show later that a consistent constraint set is satisfiable.

Definition A constraint set $\mathcal{C}$ is consistent if it is not inconsistent. $\mathcal{C}$ is inconsistent if one of the followings is true:

1) $\{V \leq \texttt{bt},\ V \leq [m : (V, \psi)]\} \subseteq \mathcal{C}$
2) $\{V \leq \texttt{bt},\ V \leq (V_0, \mathcal{M}_0) \times (V_1, \mathcal{M}_1) \to V_2\} \subseteq \mathcal{C}$
3) $\{V \leq [m : (V, \psi)],\ V \leq (V_0, \mathcal{M}_0) \times (V_1, \mathcal{M}_1) \to V_2\} \subseteq \mathcal{C}$
4) $V_F \leq [m : (V, \bullet)] \in \mathcal{C}$.

The first three rules of inconsistency make sure that a type variable cannot be both integer (or boolean) and an object type (or a function type) or be both an object type and a function type at the same time. The last rule says that within a constructor function, the type of `this` pointer cannot include a definite member since a constructor function is always invoked with an empty object substituting `this` and empty object's type cannot have definite member. This is the only rule that catches the errors of accessing undefined member. Intuitively, member access on an object introduces constraint of the form $(V, M) \leq [m : (V', \bullet)]$. If the member is not added before the access, then the closure rules will eventually generate $V_F \leq [m : (V', \bullet)]$, where V_F is the initial self type of the constructor. Otherwise, only constraints of the form $V_F \leq [m : (V', \circ)]$ may be generated.

D. Constraint solution

We define a function $\mathsf{Upper}_{\mathcal{C}}(\mathcal{M})$ to find the upper bounds to $\mathcal{M}$ variables, where $M_\top$ is the set of all member names in the program.

$$\mathsf{Upper}_{\mathcal{C}}(\mathcal{M}) = \{(\bigcap_j \texttt{mSet}_j) \cap M_\top \mid \mathcal{M} \subseteq \texttt{mSet}_j \in \mathcal{C}\}$$

We can show that $\mathsf{Upper}_{\mathcal{C}}(\mathcal{M})$ satisfies the constraints in $\mathcal{C}$ if $\mathcal{C}$ is closed and consistent. Note that it is possible that some $\mathcal{M}$ variables are not constrained if a program does not terminate and in that case, the upper bound to the $\mathcal{M}$ variables is $M_\top$. For example, the call `y.m()` shown below will cause an infinite loop. However, the program is still typable.

```
1  function f(x) {
2     this.m(0);
3     return 0;
4  }
5  Function F() {
6     this.m = f;
7  }
8  y = new F();
9  z = y.m(0);
10 z1 = y.m1;
```

We can give function f the type t_f

$$t_f = (t_{\texttt{this}}, \{\texttt{m1}\}) \times (\mathsf{int}, \emptyset) \to \mathsf{int}$$

so that when y starts with a type $(t_{\mathsf{y}}, \emptyset)$ where $t_{\mathsf{y}} = [m : (t_f, \bullet)]$, it ends with the type $(t_{\mathsf{y}}, \{\texttt{m1}\})$ after the method call. Then the statement `z1 = y.m1` becomes typable. Notice that the function f never adds a `m1` member to self but we can make any assumption for the function type as long as it doesnot violate any constraints. This typing is indeed safe because we can show that if a program terminates, then $\mathcal{M}$ variables are always bounded.

We also use $\mathsf{Upper}_{\mathcal{C}}(V)$ to obtain upper bound of a type variable V in a constraint set $\mathcal{C}$.

$$\mathsf{Upper}_{\mathcal{C}}(V) = \{W \mid (V \leq W) \in \mathcal{C}\}$$

Definition For a constraint set $\mathcal{C}$, we define its solution S (written as $\mathsf{Solution}(\mathcal{C})$) as follows:

1) $S(\mathcal{M}) = \mathsf{Upper}_{\mathcal{C}}(\mathcal{M})$;
2) $S(V) = \texttt{bt}$ if $\texttt{bt} \in \mathsf{Upper}_{\mathcal{C}}(V)$;
3) $S(V) = \mathsf{top}$ if $\mathsf{Upper}_{\mathcal{C}}(V)$ may only have variables;
4) For any other V and V', $S(V) = S(V') = t$ iff $\{V \leq V', V' \leq V\} \subseteq \mathcal{C}$, and
   a) if $(V_0, \mathcal{M}_0) \times (V_1, \mathcal{M}_1) \to V_2 \in \mathsf{Upper}_{\mathcal{C}}(V)$, then t is defined by $t = (S(V_0), S(\mathcal{M}_0)) \times (S(V_1), S(\mathcal{M}_1)) \to S(V_2)$;
   b) if $[m : \_] \in \mathsf{Upper}_{\mathcal{C}}(V)$, then $t = [m_i : (u_i, \psi_i)^{i \in N}]$, where $\forall i \in N$, $u_i = S(V')$ for some V' such that $[m_i : (V', \_)] \in \mathsf{Upper}_{\mathcal{C}}(V)$, and $\psi_i = \bullet$ if $\exists [m_i : (\_, \bullet)] \in \mathsf{Upper}_{\mathcal{C}}(V)$ and $\psi_i = \circ$ otherwise.

To find a solution S that satisfies a constraint set $\mathcal{C}$, we first find the variables that must have base types and for any other variable that only has variables in its upper bound set, we set the variable to the top type. For the rest of variables, we create equivalence partitions of these variables, where two variables V,V' are in the same partition iff $V \leq V'$ and $V' \leq V$ are in $\mathcal{C}$. For each equivalence partition, we create a unique type name t and assign it to each variable in that partition. The type names are defined by equations to associate them with function types or object types based on the type upper bounds of the corresponding variables in the constraint set. If we assign a function or object type to a type variable, we add an equation to define such type.

E. Type inference as constraint closure consistency

We show that type inference is equivalent to checking the consistency of constraint closure so that a program is typable if and only if the closure of the constraint set generated from the program is consistent. For a consistent constraint set $\mathcal{C}$, we can find a satisfiable solution as in Section IV-D.

Lemma IV.1. *If $\mathcal{C}$ is closed and consistent, then it is satisfiable.*

Lemma IV.2. *If $\mathcal{C}$ is satisfiable, then* $\mathsf{Closure}(\mathcal{C})$ *is consistent.*

We can prove Lemma IV.1 by showing that if a constraint set $\mathcal{C}$ is closed and consistent, then $\mathsf{Solution}(\mathcal{C})$ is a satisfiable solution to each kind of constraints in $\mathcal{C}$. To prove Lemma IV.2, we only need to show that if $\mathcal{C}$ is satisfiable, then its closure is also satisfiable. A satisfiable constraint set is always consistent. Theorem IV.3 concludes that our type inference algorithm is sound and complete with respect to our type system.

Theorem IV.3. *Given a program P where $E \vdash_{\mathsf{inf}} P \mid \mathcal{C}$, P is typable iff* $\mathsf{Closure}(\mathcal{C})$ *is consistent.*

V. Allow strong updates to new objects

Since we assumed that constructor functions always return new objects, constructors return singleton types. For simplicity, we do not assign singleton types to objects returned from regular functions. The meta symbol ς ranges over the singleton type names, which are distinct from the obj-type names. Each singleton type ς is defined by an equation of the form

$$\varsigma = @[m_i : (u_i, \psi_i)^{i \in 1..n}],$$

where u_i cannot be singleton types and ψ_i may be either $*$, $\bullet$, or $\circ$. We define $* \leq \bullet$ and $* \leq *$. An object of singleton type can have unrestricted extensions and its members with the label $*$ can receive strong updates. Also, given ς defined as above, $\varsigma(m_i) = (u_i, \psi_i)$ for all $i \in 1..n$ and $\varsigma(m_i) = \mathsf{undef}$ otherwise. The member m of a singleton type ς can be read if $\varsigma \leq [m : (u, \bullet)]$.

$$\frac{\varsigma(m) = (u, \psi) \quad \psi \leq \bullet}{\varsigma \leq [m : (u, \bullet)]}$$

A singleton type ς can be extended with a set of members M to form a new singleton type ς' define by the relation $\varsigma \leq_M \varsigma'$.

$$\frac{\begin{array}{c}\forall m \notin M.\ \varsigma'(m) = \varsigma(m) \\ \forall m \in M.\ \varsigma'(m) = \varsigma(m) \ \vee \\ (\varsigma'(m) = (u, *) \ \wedge \ \varsigma(m) = \mathsf{undef}) \ \vee \\ (\varsigma'(m) = (u, \bullet) \ \wedge \ \varsigma(m) = (u, \circ))\end{array}}{\varsigma' \leq_M \varsigma}$$

A. Type rules

Before introducing new type rules, we first define an operator $\downarrow$ to downgrade the singleton type ς to ς' so that some of the $*$ members in ς are definite or potential in ς' and ς' may have some additional potential members. A singleton type may become more restrictive after downgrading since some of its members may not be changed and it may be restricted in member extensions.

$$\frac{\begin{array}{c}\varsigma' = @[m_i : (u_i, \psi'_i)^{i \in 1..n}, m_j : (u_j, \circ)^{j \in n+1..m}] \\ \varsigma = @[m_i : (u_i, \psi_i)^{i \in 1..n}] \\ \forall i \in 1..n \quad \psi_i \leq \psi'_i \leq \bullet \ \vee \ \psi_i = \psi'_i = \circ\end{array}}{\varsigma \downarrow \varsigma'}$$

A singleton type ς is a subtype of an obj-type t if some definite members of ς appear as potential members in t while members labeled with $*$ in ς do not appear in t.

$$\frac{t(m) = (u, \psi') \Rightarrow (\varsigma(m) = (u, \psi) \ \wedge \ \bullet \leq \psi \leq \psi')}{\varsigma \leq t}$$

The above relations are used in a situation where an object referenced by a variable of singleton type is also referenced by a variable of obj-type. This can happen when a variable of singleton type ς is passed to a parameter of the type (t, M), or is assigned to a field of type t. To preserve type safety, we first downgrade ς to ς' so that it is a subtype of t, then extend ς' with members in M to obtain ς'', and finally replace all occurrences of ς in the type environment with ς''. These steps are included in the new rule for the judgment $\Gamma \vdash y : \tau \ \parallel \ \Gamma'$ as shown below.

$$\frac{\Gamma(y) = \varsigma \quad \varsigma \downarrow \varsigma' \quad \varsigma' \leq t \quad \varsigma'' \leq_M \varsigma'}{\Gamma \vdash y : (t, M) \ \parallel \ [\varsigma''/\varsigma]\Gamma}$$

The new rule for judgment $\Gamma; \Gamma' \vdash y : (t, M) \ \parallel \ \Gamma''$ is used in Rule (T-Invk) to check that y can be passed to the self pointer with the type (t, M) of a function. Γ is the original type environment while Γ' is the environment after checking the argument to the function parameter. Since y could also be the argument, $\Gamma'(y)$ may not be the same as $\Gamma(y)$. Thus, we need to make sure that $\Gamma(y)$ is compatible with t before checking that $\Gamma'(y)$ is also compatible.

$$\frac{\begin{array}{c}\Gamma(y) = \varsigma_1 \quad \varsigma_1 \downarrow \varsigma'_1 \quad \varsigma'_1 \leq t \\ \Gamma'(y) = \varsigma_2 \quad \varsigma_2 \downarrow \varsigma'_2 \quad \varsigma'_2 \leq t \quad \varsigma \leq_M \varsigma'_2\end{array}}{\Gamma; \Gamma' \vdash y : (t, M) \ \parallel \ [\varsigma/\varsigma_2]\Gamma'}$$

The substitution of singleton types in Γ is defined as:

$$[\varsigma'/\varsigma]\emptyset = \emptyset \qquad [\varsigma'/\varsigma](y \mapsto \varsigma, \Gamma) = y \mapsto \varsigma', [\varsigma'/\varsigma]\Gamma$$

$$[\varsigma'/\varsigma](y \mapsto \tau, \Gamma) = y \mapsto \tau, [\varsigma'/\varsigma]\Gamma$$

Consider the following example:

```
1  function F() {
2     this.a = 0;
3     this.m = f;
4  }
5  function f(y) {
6     y.a = 1;
7     y.b = 2;
8     return y;
9  }
10 x = new F();
11 x1 = new F();
12 x2 = x;
13 z = x1.m(x);
14 x2.a = true          // type error
15 z1 = z.a;
```

The type of x at line 10 is ς_{x} and it becomes ς''_{x} after line 13.

$$\begin{aligned}
\varsigma_{\mathsf{x}} &= @[\mathsf{a} : (\mathsf{int}, *), \mathtt{m} : (t_f, *)] \\
\varsigma'_{\mathsf{x}} &= @[\mathsf{a} : (\mathsf{int}, \bullet), \mathtt{m} : (t_f, *), \mathsf{b} : (\mathsf{int}, \circ)] \\
\varsigma''_{\mathsf{x}} &= @[\mathsf{a} : (\mathsf{int}, \bullet), \mathtt{m} : (t_f, *), \mathsf{b} : (\mathsf{int}, \bullet)] \\
t_f &= (\mathsf{top}, \emptyset) \times (t_{\mathsf{y}}, \{\mathsf{a}, \mathsf{b}\}) \to t_{\mathsf{z}} \\
t_{\mathsf{y}} &= [\mathsf{a} : (\mathsf{int}, \circ), \mathsf{b} : (\mathsf{int}, \circ)],\ t_{\mathsf{z}} = [\mathsf{a} : (\mathsf{int}, \bullet)]
\end{aligned}$$

Before the variable x is passed to the parameter y of the function f, we downgrade the type ς_{x} to ς'_{x} so that $\varsigma_{\mathsf{x}} \leq t_{\mathsf{y}}$, where t_{y} is the type of the parameter y. The subtyping relation $\varsigma'_{\mathsf{x}} \leq t_{\mathsf{y}}$ guarantees that the $*$ members of ς'_{x} do not appear in t_{y} so that strong updates to these members are not visible through t_{y}. Specifically, the member a of ς'_{x} is definite so that it cannot have strong updates. This is necessary since the variable z is an alias of y while a is a definite member in t_{z} – the type of z. Lastly, we extend ς'_{x} with the set $\{\mathsf{a}, \mathsf{b}\}$ to obtain ς''_{x} to replace ς_{x} as the type of x and x2.

To see why the type substitution is necessary, consider the previous example where the variable x2 and x are aliases of each other and they both have the type ς_{x}. If the type of x2 remains ς_{x} after the call, then x2.a = true will change the member z.a to boolean, since x2 and z refer to the same object. This would be inconsistent with the type of z.

Finally, we are ready to define the type rule for constructors.

$$\frac{\Gamma[\mathtt{this} \mapsto \varsigma, x_p \mapsto (u_{arg}, \emptyset)] \vdash s \ \parallel\ \Gamma' \qquad \Gamma'(\mathtt{this}) \downarrow \varsigma_{res} \quad \varsigma = @[\,] \quad \Gamma(F) = \Gamma'(x_p) \to \varsigma_{res}}{\Gamma \vdash \mathsf{function}\ F(x_p)\{s\}} \ \text{T-Ctr}$$

What is different is that the type of this pointer in the constructor function is initially assigned a singleton type that has no members and the type of this may be replaced by other singleton type after the function body is evaluated.

We also modify the type rules for updates and new statements, in Figure 6. For $x = \mathsf{new}\ F(z)$, we create a singleton type that is downgraded from the return type of F for x. For $y.m_j = z$, if y has a singleton type ς, we extend that type with the member m_j to obtain ς' and let y map to ς' in the new type environment. The update is a strong update or an extension if either m_j has the label $*$ or is not defined in ς.

$$\frac{\begin{array}{rcl}\Gamma(y) = \varsigma & \Rightarrow & \varsigma' \leq_{(m,u)} \varsigma \quad \Gamma' = [\varsigma'/\varsigma]\Gamma \\ \Gamma(y) = (t, M) & \Rightarrow & \Gamma(y) \leq [m : (u, \circ)] \\ & & M' \subseteq M \cup \{m\} \\ & & \Gamma' = \Gamma[y \mapsto (t, M']\end{array} \qquad \Gamma' \vdash z : (u, \emptyset) \ \parallel\ \Gamma''}{\Gamma \vdash y.m = z \ \parallel\ \Gamma''} \ \text{T-Upd}$$

$$\frac{\Gamma(F) = \tau_{arg} \to \varsigma_{res} \quad \Gamma \vdash z : \tau_{arg} \ \parallel\ \Gamma' \quad \varsigma_{res} \downarrow \varsigma}{\Gamma \vdash x = \mathsf{new}\ F(z) \ \parallel\ \Gamma'[x \mapsto \varsigma]} \ \text{T-New}$$

Fig. 6. New type rules for statements

The relation $\varsigma' \leq_{(m,\tau)} \varsigma$ describes the update/extension of a singleton type ς with member m of type u that results in ς'. If m is not defined or it is labeled with $*$ in ς, then the object can receive strong update.

$$\frac{\varsigma(m) = \mathsf{undef} \ \vee\ \varsigma(m) = (\_, *) \quad \varsigma' = \varsigma[m \mapsto (u, *)]}{\varsigma' \leq_{(m,u)} \varsigma}$$

$$\frac{\varsigma'(m) = (u, \psi) \quad \bullet \leq \psi \quad \varsigma' \leq_{\{m\}} \varsigma}{\varsigma' \leq_{(m,u)} \varsigma}$$

Consider the previous example, the types of this at line 2 – $\varsigma_{\mathtt{this}}$ is changed to $\varsigma'_{\mathtt{this}}$ at line 3 with $\varsigma'_{\mathtt{this}} \leq_{(\mathtt{m}, t_f)} \varsigma_{\mathtt{this}}$.

$$\begin{aligned}
\varsigma_{\mathtt{this}} &= @[\mathsf{a} : (\mathsf{int}, *)] \\
\varsigma'_{\mathtt{this}} &= @[\mathsf{a} : (\mathsf{int}, *), \mathtt{m} : (t_f, *)]
\end{aligned}$$

In the branch statements, a variable with a different singleton type in each branch will have to be assigned a new objtype since a singleton type can only be used for one object. The exception is the type of this pointer of a constructor, which has singleton type and is immutable. So if this has different types in the branch statements, we can merge the two singleton types into one singleton type without violating type safety. The details of the merge rules for singleton types are in the appendix.

VI. Related work

Our work is similar to the type inference system of Anderson et al. [1] on 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. We follow their lead in using method labels to denote members of an object as being definite or potential. In addition to explicit member extension, we also allow implicit extension through function parameter and self pointer. We distinguish constructor functions from regular functions in that constructors are used in new expressions that always return new objects. This distinction allows new objects to have strong updates and unrestricted extensions.

Also related is the work of Gianantonio et al. [4] on lambda calculus of objects with self-inflicted extension. Instead of using labels, they separate potential and definite 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 define a type construct to recursively encode member extension information for methods. In comparison, we extend each function type with a set of members that are added to `this` in the function body. They 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. We allow newly created objects to have singleton types similar to their pro-types. The difference is that an object of singleton type do not lose the ability of having strong updates even after it is assigned to parameters or fields of obj-type.

Recency types of Heidegger and Thiemann [2] 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 the singleton type has to be promoted to the corresponding summary type when the next object is allocated at the same location. Objects of singleton types can receive strong updates for adding new members or even changing the types of existing members. Objects of summary types can no longer be extended. Moreover, they support prototypes and have implemented a constraint-based type inference algorithm. In their formalism, abstract locations are assigned to new expressions that return empty objects and if a function will extend its parameter, the parameter type needs to be singleton type. This may present some challenges for supporting explicit extension. For example, the `setter` function discussed earlier may be a member of two different constructors. In order to extend `this` pointer, `setter`'s receiver type needs to be a singleton type, which forces the two constructors to return objects of the same singleton type. This can be limiting as the two constructors are forced to have the set of members of the same types. In comparison to our work, recency type allows singleton types to be in the object fields and function parameters, though it is more complex and does not allow extension after an object loses its recency.

Jensen et al. [6] 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 [7] 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.

DRuby [8] 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.

As for type inference for object-based languages, Palsberg developed efficient type inference algorithms [9] with recursive types and subtyping for Abadi Cardelli object calculus [10], 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 [11] and supporting record concatenation [12].

VII. Conclusion

We have presented a 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 extension as well as implicit extension of objects by invoking their methods. We have proved that the type inference algorithm is sound and complete so that a program is typable if and only if we can infer its types. We also included an extension to allow strong updates to new objects.

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 [13]. However, our system is lack of many important features found in real world JavaScript programs such as prototypes, variadic functions, eval function, member deletion, and objects as associative arrays. For future work, we would like to have more flexible subtyping rules and to allow parametric polymorphism.

References

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

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

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

[4] 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)*, 1998, pp. 166–178.

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

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

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

[8] 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*, 2009, pp. 1859–1866.

[9] J. Palsberg, "Efficient Inference of Object Types," *Inf. Comput.*, vol. 123, no. 2, pp. 198–209, 1995.

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

[11] J. Palsberg, T. Zhao, and T. Jim, "Automatic discovery of covariant read-only fields," *ACM Trans. Program. Lang. Syst.*, vol. 27, no. 1, pp. 126–162, 2005.

[12] J. Palsberg and T. Zhao, "Type Inference for Record Concatenation and Subtyping," *Inf. Comput.*, vol. 189, no. 1, pp. 54–86, 2004.

[13] 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)*, Jun. 2010.

APPENDIX

A. A type inference example

In this section, we show some of the type inference steps for the example in Figure 1. We simplify the example slightly and reproduce it below.

```
1  function Form(a) {
2     this.set = setter;
3  }
4  function setter(b) {
5     this.handle = b;
6     return 0;
7  }
8  function handler(c) {
9     return 0;
10 }
11 // main
12 x = new Form(1);
13 y = x.set(handler);
14 z = x.handle(1);
```

We first show the constraints generated from each function in Figure 7, where we choose type variable names based on the names of the corresponding variable. For example, the type variable for function `setter` is V_{Setter}. The exception is V_{Form}, which is the type variable corresponding to the initial type of `this` pointer in the constructor function Form. Type variables for other types are sequential numbered to avoid collision.

Function	Generated constraints
Form	$(V_{Form}, \mathcal{M}_\emptyset) \leq [\texttt{set} : (V_{set}, \circ)]$ $V_{setter} \leq V_{set} \quad (V_{Form}, \mathcal{M}_{Form}) \leq V_x$ $\mathcal{M}_{Form} \subseteq \mathcal{M}_\emptyset \cup \mathcal{M}_{set} \quad \texttt{set} \in \mathcal{M}_{set}$
setter	$V_{setter} \leq (V_{sTh}, \mathcal{M}_{sTh}) \times (V_b, \mathcal{M}_\emptyset) \to V_{sRet}$ $(V_{sTh}, \mathcal{M}_\emptyset) \leq [\texttt{handle} : (V_{handle}, \circ)]$ $V_b \leq V_{handle} \quad \texttt{handle} \in \mathcal{M}_{handle}$ $\mathcal{M}_{sTh} \subseteq \mathcal{M}_\emptyset \cup \mathcal{M}_{handle} \quad \mathsf{int} \leq V_{sRet}$
handler	$V_{hdler} \leq (V_{hTh}, \mathcal{M}_\emptyset) \times (V_c, \mathcal{M}_\emptyset) \to V_{hRet}$ $\mathsf{int} \leq V_{hRet}$
main	$\mathsf{int} \leq V_a$ $(V_x, \mathcal{M}_\emptyset) \leq [\texttt{set} : (V_{x.set}, \bullet)]$ $V_{x.set} \leq (V_{xTh}, \mathcal{M}_{xTh}) \times (V_{xb}, \mathcal{M}_{xb}) \to V_{xRet}$ $V_x \leq V_{xTh} \quad V_{hdler} \leq V_{xb}$ $\mathcal{M}_x \subseteq \mathcal{M}_\emptyset \cup \mathcal{M}_{xTh}$ $(V_x, \mathcal{M}_x) \leq [\texttt{handle} : (V_{x.hd}, \bullet)]$ $V_{x.hd} \leq (V_{dTh}, \mathcal{M}_{dTh}) \times (V_{dc}, \mathcal{M}_{dc}) \to V_{dRet}$ $(V_x, \mathcal{M}_x) \leq V_{dTh} \quad \mathsf{int} \leq V_{dc}$

Fig. 7. Generated constraints

After applying the closure rules, we collect the types in the upper bound of each variable, most of which are shown in Figure 8. We can verify that the closure of the original constraints set is consistent. In particular, the upper bound of V_{Form} is $\{[\texttt{set} : (V_{set}, \circ)], [\texttt{set} : (V_{x.set}, \circ)], [\texttt{handle} : (V_{x.hd}, \circ)]\}, [\texttt{handle} : (V_{handle}, \circ]$ and satisfies the consistency rule that it may not contain object types with definite members.

Types	Type upper bound
V_a	int
V_{set}	$V_{x.set}$, $(V_{xTh}, \mathcal{M}_{xTh}) \times (V_{xb}, \mathcal{M}_{xb}) \to V_{xRt}$
V_{Form}	$[\texttt{set} : (V_{set}, \circ)]$, $[\texttt{set} : (V_{x.set}, \circ)]$, $[\texttt{handle} : (V_{x.hd}, \circ)], [\texttt{handle} : (V_{handle}, \circ]\}$
V_{setter}	$(V_{sTh}, \mathcal{M}_{sTh}) \times (V_b, \mathcal{M}_\emptyset) \to V_{sRet}$, V_{set}, $V_{x.set}$, $(V_{xTh}, \mathcal{M}_{xTh}) \times (V_{xb}, \mathcal{M}_{xb}) \to V_{xRet}$
V_{sTh}	$[\texttt{handle} : (V_{handle}, \circ)]$, V_{xTh}
V_{sRet}	int, V_{xRet}
V_b	V_{handle}, V_{xb}, $V_{x.hd}$, $(V_{dTh}, \mathcal{M}_{dTh}) \times (V_{dc}, \mathcal{M}_{dc}) \to V_{dRet}$
V_{handle}	$V_{x.hd}$, $(V_{dTh}, \mathcal{M}_{dTh}) \times (V_{dc}, \mathcal{M}_{dc}) \to V_{dRet}$
V_{hdler}	$(V_{hTh}, \mathcal{M}_\emptyset) \times (V_c, \mathcal{M}_\emptyset) \to V_{hRet}$, $(V_{dTh}, \mathcal{M}_{dTh}) \times (V_{dc}, \mathcal{M}_{dc}) \to V_{dRet}$, V_{xb}, V_b, V_{handle}, $V_{x.hd}$
V_c	V_{dc}, int
V_{hTh}	V_{dTh}
V_{hRet}	int, V_{dRet}
V_x	$[\texttt{set} : (V_{x.set}, \bullet)]$, V_{xTh}, V_{sTh}, $[\texttt{handle} : (V_{handle}, \circ)]$, $[\texttt{handle} : (V_{x.hd}, \circ)]$
$V_{x.set}$	$(V_{xTh}, \mathcal{M}_{xTh}) \times (V_{xb}, \mathcal{M}_{xb}) \to V_{xRet}$, V_{set}
V_{xTh}	V_{sTh}, $[\texttt{handle} : (V_{handle}, \circ)]$
V_{xb}	V_b, V_{handle}, $V_{x.hd}$, $(V_{dTh}, \mathcal{M}_{dTh}) \times (V_{dc}, \mathcal{M}_{dc}) \to V_{dRet}$
V_{xRet}	V_{sRet}, int
$V_{x.hd}$	$(V_{dTh}, \mathcal{M}_{dTh}) \times (V_{dc}, \mathcal{M}_{dc}) \to V_{dRet}$, V_{handle}
V_{dTh}	V_{hTh}
V_{dc}	int, V_c
V_{dRet}	int, V_{hRet}

Fig. 8. Type upper bounds of each type variable

Closure also generates some constraints for $\mathcal{M}$ variables with clear solutions.

$$\mathcal{M}_\emptyset \subseteq \emptyset$$
$$\mathcal{M}_{sTh} = \mathcal{M}_{xTh} \quad \mathcal{M}_{xb} = \mathcal{M}_\emptyset \quad \mathcal{M}_{dc} = \mathcal{M}_\emptyset \quad \mathcal{M}_{dTh} = \mathcal{M}_\emptyset$$
$$\mathcal{M}_{set} \subseteq \{\texttt{set}\} \quad \mathcal{M}_{handle} \subseteq \{\texttt{handle}\} \quad \mathcal{M}_x \subseteq \mathcal{M}_\emptyset \cup \mathcal{M}_{xTh}$$
$$\mathcal{M}_{Form} \subseteq \mathcal{M}_\emptyset \cup \mathcal{M}_{set} \quad \mathcal{M}_{sTh} \subseteq \mathcal{M}_\emptyset \cup \mathcal{M}_{handle}$$

From the type upper bounds, we can obtain solutions to each type. The solutions to most of the variables and functions are shown in Figure 9, where the variable x has a type before and after the call to the set method.

Names	Corresponding types
Form	$(\mathsf{int}, \emptyset) \to t_x$
a	int
x	$t_x = [\mathtt{set} : (t_{set}, \bullet),\ \mathtt{handle} : (t_{handle}, \circ)]$
	$t_{set} = (t_{sTh}, \{\mathtt{handle}\}) \times t_b \to \mathsf{int}$
setter	$t_{setter} = (t_{sTh}, \{\mathtt{handle}\}) \times (t_b, \emptyset) \to \mathsf{int}$
	$t_{sTh} = [\mathtt{handle} : (t_{handle}, \circ)]$
	$t_{handle} = (\mathsf{top}, \emptyset) \times (\mathsf{int}, \emptyset) \to \mathsf{int}$
b	$t_b = (\mathsf{top}, \emptyset) \times (\mathsf{int}, \emptyset) \to \mathsf{int}$
handler	$t_{handler} = (\mathsf{top}, \emptyset) \times (\mathsf{int}, \emptyset) \to \mathsf{int}$
c	int
x	$(t_x, \{\mathtt{handle}\})$
y	int
z	int

Fig. 9. Constraint solution

B. Operational semantics

We define a big-step semantics for our language in Figure 10. 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.

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

Also, $\chi(n) = n$ for any integer n and $\chi(f) = f$ for any function name f. 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 $j \in 1..n$.

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

$$\frac{x \notin \mathsf{dom}(\chi)}{H, \chi, \mathsf{var}\ x \rightsquigarrow H, \chi[x \mapsto \mathsf{null}]} \text{ R-Dec}$$

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

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

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

$$\frac{\begin{array}{c}H(\chi(y)) = o \\ H' = H(\chi(y) \mapsto o[m_j \mapsto \chi(z)])\end{array}}{H, \chi, y.m_j = z \rightsquigarrow H', \chi} \text{ R-Upd}$$

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

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

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

Fig. 10. Operational semantics where the reduction rules of statements assume an implicit function table FT that maps each function/constructor name to its declaration.

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.

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

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 using an undefined function/constructor name. We use a special configuration error to denote the result of the computation as shown in Figure 11. 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 12. We tolerate this type of error.

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

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

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

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

Fig. 11. Error of accessing undefined members or functions

C. Type soundness

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

The judgment $\Sigma, \Gamma \vdash v : u$ asserts that the value v is well-typed with the type u.

$$\Sigma, \Gamma \vdash n : \mathsf{int} \quad \Sigma, \Gamma \vdash b : \mathsf{bool} \quad \Sigma, \Gamma \vdash \mathsf{null} : t$$

$$\frac{\Sigma(\iota) \leq t}{\Sigma, \Gamma \vdash \iota : t} \qquad \frac{\Gamma(f) \leq t}{\Sigma, \Gamma \vdash f : t} \qquad \frac{\Sigma, \Gamma \vdash v : u \quad u \leq \tau}{\Sigma, \Gamma \vdash v : \tau}$$

$$\mathsf{int} \leq (\mathsf{int}, \_) \quad \mathsf{bool} \leq (\mathsf{bool}, \_) \quad t_f \leq (t_f, \_)$$

where t_f represents function types.

$$\frac{\begin{array}{c} \forall m.\ t'(m) = (u, \psi') \Rightarrow t(m) = (u, \psi) \ \wedge \\ (\psi' = \bullet \ \vee \ m \in M) \Rightarrow \psi = \bullet \end{array}}{t \leq (t', M)}$$

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

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

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

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

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

Fig. 12. Null pointer exception

For an object to be well-typed, each of the object's member value must be well-typed and it must be a definite member in the object's type. The judgment $\Sigma, \Gamma \vdash o : t$ asserts that the object o is well-typed with the type t.

$$\frac{\forall m.\ t(m) = (u, \bullet) \Rightarrow \Sigma, \Gamma \vdash o(m) : u}{\Sigma, \Gamma \vdash o : t}$$

We define a well-formed type τ (written as $\vdash \tau$) as below.

$$\frac{\forall m \in M.\ u \leq [m : (\_, \circ)]}{\vdash (u, M)}$$

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

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

The judgment $\Sigma, \Gamma \vdash H, \chi$ says that the heap H and stack χ are well-formed with respect to the environment Σ and Γ. 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 object in H and each variable in χ must be well-typed. Each function/constructor declaration in χ is well-typed with the environment $\Gamma_{\mathtt{init}}$, which is defined as the environment that maps function/constructor names to their types.

From the typing rules, we can show that if a well-typed function f has the type $(t, M_{\mathtt{this}}) \times (u, M_{arg}) \ \rightarrow \ u_{res}$, then $M_{\mathtt{this}}$ and M_{arg} correctly identify the members of the receiver and the argument that are added (or updated). Based

on this result, we can show 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* $\Sigma, \Gamma \vdash H, \chi$ *and* $\Gamma \vdash s \parallel \Gamma'$*, then* $H, \chi, s \not\rightsquigarrow$ error*, and if* $H, \chi, s \rightsquigarrow H', \chi'$*, then* $\exists\ \Sigma'$ *such that* $\Sigma', \Gamma' \vdash H', \chi'$.

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

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

D. Type inference

The type inference rules shown in Figure 13 mirror the type rules. The inference rules uses the judgment of the form $E \vdash_{\mathsf{inf}} z : (V, M) \parallel E' \mid \mathcal{C}$ to infer a constraint set $\mathcal{C}$ when passing the variable z to a parameter of the type (V, M).

$$\frac{z = n \mid b \mid f}{E \vdash_{\mathsf{inf}} z : (V, \mathcal{M}) \parallel E \mid \{E(z) \leq V\}}$$

$$\frac{E(y) = (V_y, \mathcal{M}_y) \quad \mathcal{M}' \text{ fresh} \quad \mathcal{C} = \{E(y) \leq V, \mathcal{M}' \subseteq \mathcal{M}_y \cup \mathcal{M}\}}{E \vdash_{\mathsf{inf}} y : (V, \mathcal{M}) \parallel E[y \mapsto (V_y, \mathcal{M}')] \mid \mathcal{C}}$$

$$\frac{E'(y) = (V_y, \mathcal{M}_y) \quad \mathcal{M}' \text{ fresh} \quad \mathcal{C} = \{E(y) \leq V, \mathcal{M}' \subseteq \mathcal{M}_y \cup \mathcal{M}\}}{E; E' \vdash_{\mathsf{inf}} y : (V, \mathcal{M}) \parallel E'[y \mapsto (V_y, \mathcal{M}')] \mid \mathcal{C}}$$

The predicate $\mathtt{merge}(E_1, E_2, E, \mathcal{C})$ merges the environments E_1 and E_2 to E and results in a constraint set $\mathcal{C}$.

$$\mathtt{merge}(E, E, E, \emptyset) \qquad \frac{\mathtt{merge}(E_1, E_2, E, \mathcal{C})}{\mathtt{merge}(E_2, E_1, E, \mathcal{C})}$$

$$\frac{y \notin \mathsf{dom}(E_2) \quad \mathtt{merge}(E_1, E_2, E, \mathcal{C})}{\mathtt{merge}((y \mapsto \_, E_1), E_2, E, \mathcal{C})}$$

$$\frac{\mathcal{M} \text{ fresh} \quad \mathtt{merge}(E_1, E_2, E', \mathcal{C}') \quad \mathcal{C} = \mathcal{C}' \cup \{\mathcal{M} \subseteq \mathcal{M}_1 \cap \mathcal{M}_2\} \quad E = y \mapsto (V, \mathcal{M}), E'}{\mathtt{merge}((y \mapsto (V, \mathcal{M}_1), E_1), (y \mapsto (V, \mathcal{M}_2), E_2), E, \mathcal{C})}$$

$$\frac{V \text{ fresh} \quad V_1 \neq V_2 \quad \mathtt{merge}(E_1, E_2, E', \mathcal{C}') \quad \mathcal{C} = \mathcal{C}' \cup \{(V_1, \mathcal{M}_1) \leq V, (V_2, \mathcal{M}_2) \leq V\} \quad E = y \mapsto (V, \emptyset), E'}{\mathtt{merge}((y \mapsto (V_1, \mathcal{M}_1), E_1), (y \mapsto (V_2, \mathcal{M}_2), E_2), E, \mathcal{C})}$$

E. Allow strong update to new objects

For the *if* statement, we need to define some additional rules to merge type environments of each branch. A variable mapped to the same singleton type still has that type in the merged environment. If a variable y has different singleton types or a singleton type in one branch and an obj-type in the other branch, then we let y has an obj-type in the merged environment. The singleton types have to be downgraded to be the subtypes of the final type of y.

$$\frac{\forall i = 1, 2.\ \varsigma_i \downarrow \varsigma_i' \quad \varsigma_i' \leq t \quad \Gamma_i' = [\varsigma_1'/\varsigma_1, \varsigma_2'/\varsigma_2]\Gamma_i \quad \varsigma_1 \neq \varsigma_2 \quad \mathtt{merge}(\Gamma_1', \Gamma_2', \Gamma)}{\mathtt{merge}((y \mapsto \varsigma_1, \Gamma_1), (y \mapsto \varsigma_2, \Gamma_2), (y \mapsto (t, \emptyset), \Gamma))}$$

$$\frac{E \vdash_{\mathsf{inf}} Fn_i \mid \mathcal{C}_i\ \forall i \in 1..n \quad E \vdash_{\mathsf{inf}} s \parallel E' \mid \mathcal{C}_0 \quad \mathcal{C} = \{\mathcal{M}_\emptyset \subseteq \emptyset, V_{\mathsf{int}} \leq \mathsf{int}, V_{\mathsf{bool}} \leq \mathsf{bool}\}}{E \vdash_{\mathsf{inf}} Fn_i^{i \in 1..n}\ s \mid \bigcup_{i \in 0..n} \mathcal{C}_i \cup \mathcal{C}}$$

$$\frac{V_{\mathtt{this}}, V_{arg}, V_{res} \text{ fresh} \quad E[x \mapsto (V_{arg}, \mathcal{M}_\emptyset), \mathtt{this} \mapsto (V_{\mathtt{this}}, \mathcal{M}_\emptyset)] \vdash_{\mathsf{inf}} s \parallel E' \mid \mathcal{C}' \quad \mathcal{C}'' = \mathcal{C}' \cup \{E'(z) \leq V_{res}\} \quad \mathcal{C} = \mathcal{C}'' \cup \{E(f) \leq (E'(\mathtt{this}) \times E'(x) \to V_{res}\}}{E \vdash_{\mathsf{inf}} \mathsf{function}\ f(x)\{s; \mathsf{return}\ z\} \mid \mathcal{C}}$$

$$\frac{E[x \mapsto (V_{arg}, \mathcal{M}_\emptyset), \mathtt{this} \mapsto (V_F, \mathcal{M}_\emptyset)] \vdash_{\mathsf{inf}} s \parallel E' \mid \mathcal{C}' \quad E'(x) = (V_{arg}, \mathcal{M}_{arg}) \quad E(F) = (V_{arg}, \mathcal{M}) \to V_{res} \quad \mathcal{C} = \mathcal{C}' \cup \{E'(\mathtt{this}) \leq V_{res}, \mathcal{M} = \mathcal{M}_{arg}\}}{E \vdash_{\mathsf{inf}} \mathsf{function}\ F(x)\{s\} \mid \mathcal{C}}$$

$$\frac{V \text{ fresh} \quad E' = E[x \mapsto (V, \mathcal{M}_\emptyset)]}{E \vdash_{\mathsf{inf}} \mathsf{var}\ x \parallel E' \mid \emptyset}$$

$$E \vdash_{\mathsf{inf}} x = z \parallel E[x \mapsto E(z)] \mid \emptyset$$

$$\frac{V_{y.m}, \mathcal{M}_y', \mathcal{M} \text{ fresh} \quad E(y) = (V_y, \mathcal{M}_y) \quad \mathcal{C} = \{E(y) \leq [m : (V_{y.m}, \circ)], E(z) \leq V_{y.m}, \mathcal{M}_y' \subseteq \mathcal{M}_y \cup \mathcal{M}, \mathcal{M} \subseteq \{m\}\} \quad E' = E[y \mapsto (V_y, \mathcal{M}_y')]}{E \vdash_{\mathsf{inf}} y.m = z \parallel E'' \mid \mathcal{C}}$$

$$\frac{V_{y.m} \text{ fresh} \quad \mathcal{C} = \{E(y) \leq [m : (V_{y.m}, \bullet)]\} \quad E' = E[x \mapsto (V_{y.m}, \mathcal{M}_\emptyset)]}{E \vdash_{\mathsf{inf}} x = y.m \parallel E' \mid \mathcal{C}}$$

$$\frac{V_{y.m}, V_{\mathtt{this}}, V_{arg}, V_{res}, \mathcal{M}_{\mathtt{this}}, \mathcal{M}_{arg} \text{ fresh} \quad E \vdash z : (V_{arg}, \mathcal{M}_{arg}) \parallel E' \mid \mathcal{C}' \quad E; E' \vdash y : (V_{\mathtt{this}}, \mathcal{M}_{\mathtt{this}}) \parallel E'' \mid \mathcal{C}'' \quad \mathcal{C} = \{E(y) \leq [m : (V_{y.m}, \bullet)], V_{y.m} \leq (V_{\mathtt{this}}, \mathcal{M}_{\mathtt{this}}) \times (V_{arg}, \mathcal{M}_{arg}) \to V_{res}\}}{E \vdash_{\mathsf{inf}} x = y.m(z) \parallel E''[x \mapsto (V_{res}, \mathcal{M}_\emptyset)] \mid \mathcal{C} \cup \mathcal{C}' \cup \mathcal{C}''}$$

$$\frac{E(F) = (V_{arg}, \mathcal{M}_{arg}) \to V_{res} \quad E \vdash_{\mathsf{inf}} z : (V_{arg}, \mathcal{M}_{arg}) \parallel E' \mid \mathcal{C} \quad E'' = E[x \mapsto (V_{res}, \mathcal{M}_\emptyset)]}{E \vdash_{\mathsf{inf}} x = \mathsf{new}\ F(z) \parallel E' \mid \mathcal{C}}$$

$$\frac{E \vdash_{\mathsf{inf}} s \parallel E' \mid \mathcal{C} \quad E' \vdash_{\mathsf{inf}} s' \parallel E'' \mid \mathcal{C}'}{E \vdash_{\mathsf{inf}} s; s' \parallel E'' \mid \mathcal{C} \cup \mathcal{C}'}$$

$$\frac{E \vdash_{\mathsf{inf}} s_1 \parallel E_1 \mid \mathcal{C}_1 \quad E \vdash_{\mathsf{inf}} s_2 \parallel E_2 \mid \mathcal{C}_2 \quad E(z) \leq \mathsf{bool} \quad \mathtt{merge}(E_1, E_2, E', \mathcal{C})}{E \vdash_{\mathsf{inf}} \mathtt{if}(z)\{s_1\}\mathtt{else}\{s_2\} \parallel E' \mid \mathcal{C}_1 \cup \mathcal{C}_2 \cup \mathcal{C}}$$

Fig. 13. Inference rules to generate constraints from a program

$$\frac{\varsigma \downarrow \varsigma' \quad \varsigma' \le t' \quad (t, M) \le t' \quad \forall i = 1, 2.\ \Gamma'_i = [\varsigma'/\varsigma]\Gamma_i \quad \texttt{merge}(\Gamma'_1, \Gamma'_2, \Gamma)}{\texttt{merge}((y \mapsto \varsigma, \Gamma_1), (y \mapsto (t, M), \Gamma_2), (y \mapsto (t', \emptyset), \Gamma))}$$

We treat `this` variable of constructors differently since it is immutable and always has singleton type. If `this` has two different singleton types in the branches of an *if* statement, then we require these two types to be downgraded to the same singleton type.

$$\frac{\varsigma_1, \varsigma_2 \downarrow \varsigma \quad \forall i = 1, 2.\ \Gamma'_i = [\varsigma/\varsigma_1, \varsigma/\varsigma_2]\Gamma_i \quad \texttt{merge}(\Gamma'_1, \Gamma'_2, \Gamma)}{\texttt{merge}((\texttt{this} \mapsto \varsigma_1, \Gamma_1), (\texttt{this} \mapsto \varsigma_2, \Gamma_2), (\texttt{this} \mapsto \varsigma, \Gamma))}$$

where $\varsigma_1, \varsigma_2 \downarrow \varsigma$ downgrades ς_1 and ς_2 to a singleton type ς.

$$\frac{\begin{array}{c}\forall m.\ \varsigma(m) = (\_, *) \Leftrightarrow \varsigma_1(m) = (\_, *), \varsigma_2(m) = (\_, *) \\ \forall i = 1, 2. \\ \begin{array}{rcl} \varsigma(m) = (u, \_), \varsigma_i(m) = (u_i, \_) & \Rightarrow & u = u_i \\ \varsigma_i(m) = (\_, \psi_i), \bullet \le \psi_i & \Rightarrow & \varsigma(m) = (\_, \psi), \bullet \le \psi \\ \varsigma(m) = (\_, \bullet) & \Rightarrow & \varsigma_i(m) = (\_, \psi_i), \psi_i \le \bullet \end{array}\end{array}}{\varsigma_1, \varsigma_2 \downarrow \varsigma}$$

1) Type soundness: To prove type soundness, we modify the program invariant slightly. The main change is that if two singleton-type variables hold the same object, then they must have the same type.

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

Also, the environment Σ now maps each object label ι to a singleton type ς. Correspondingly, we define

$$\frac{\forall m.\ \varsigma \le [m : (u, \bullet)] \Rightarrow \Sigma, \Gamma \vdash o(m) : u}{\Sigma, \Gamma \vdash o : \varsigma}$$

The judgment $\Sigma, \Gamma \vdash o : \varsigma$ says that each definite or $*$ member of ς is defined in o, and ς may have some potential members not yet defined in o. In addition, we define a type rule to allow object labels to have singleton types and obj-types.

$$\frac{\Sigma(\iota) \le \varsigma}{\Sigma, \Gamma \vdash \iota : \varsigma} \qquad \frac{\Sigma(\iota) \le t}{\Sigma, \Gamma \vdash \iota : t}$$

A singleton type ς is a subtype of ς' if they have the same set of $*$ members but some of the definite members in ς are marked as potential in ς'.

$$\frac{\begin{array}{c}\varsigma(m) = (\tau, \psi) \Leftrightarrow \\ \varsigma'(m) = (\tau, \psi') \ \wedge\ (\psi = \psi' \ \vee\ \bullet \le \psi \le \psi')\end{array}}{\varsigma \le \varsigma'}$$

$$\frac{E[x_p \mapsto (V_{arg}, \mathcal{M}_\emptyset), \texttt{this} \mapsto \mathcal{V}_F] \vdash_{\mathsf{inf}} s \ \parallel\ E' \mid \mathcal{C} \quad E(F) = (V_{arg}, \mathcal{M}) \to \mathcal{V}_{res} \quad E'(x_p) = (V_{arg}, \mathcal{M}_{arg}) \quad \mathcal{C}' = \{E'(\texttt{this}) \downarrow \mathcal{V}_{res}, \mathcal{M} = \mathcal{M}_{arg}, \mathcal{V}_F = @[\,]\}}{E \vdash_{\mathsf{inf}} \mathsf{function}\ F(x)\{s\} \mid \mathcal{C} \cup \mathcal{C}'}$$

$$\frac{\begin{array}{c}\mathcal{V}'_y, \mathcal{M}'_y, \mathcal{M}, V \text{ fresh} \\ \begin{array}{rcl} E(y) = \mathcal{V}_y & \Rightarrow & \mathcal{C}' = \{\mathcal{V}'_y \le_{(m,V)} \mathcal{V}_y\} \\ & & E' = [\mathcal{V}'_y/\mathcal{V}_y]E \\ E(y) = (V_y, \mathcal{M}_y) & \Rightarrow & E' = E[y \mapsto (V_y, \mathcal{M}'_y)] \\ & & \mathcal{C}' = \{V_y \le [m : (V, \circ)], \\ & & \mathcal{M} \subseteq \{m\}, \mathcal{M}'_y \subseteq \mathcal{M}_y \cup \mathcal{M}\} \end{array} \\ E' \vdash_{\mathsf{inf}} z : (V, \mathcal{M}_\emptyset) \ \parallel\ E'' \mid \mathcal{C}\end{array}}{E \vdash_{\mathsf{inf}} y.m = z \ \parallel\ E'' \mid \mathcal{C} \cup \mathcal{C}'}$$

$$\frac{\mathcal{V} \text{ fresh} \quad E(F) = (V_{arg}, \mathcal{M}_{arg}) \to \mathcal{V}_{res} \quad E \vdash_{\mathsf{inf}} z : (V_{arg}, \mathcal{M}_{arg}) \ \parallel\ E' \mid \mathcal{C}}{E \vdash_{\mathsf{inf}} x = \mathsf{new}\ F(z) \ \parallel\ E'[x \mapsto \mathcal{V}] \mid \mathcal{C} \cup \{\mathcal{V}_{res} \downarrow \mathcal{V}\}}$$

Fig. 14. New type inference rules

2) Type inference: We need to modify the type inference rules for constructor function, new statement, and update in a way similar to the type rules as in Figure 14. We use the variable $\mathcal{V}$ to represent singleton types while V still represents obj-types.

We add some inference rules for judgments of the form $E \vdash_{\mathsf{inf}} y : (V, \mathcal{M}) \ \parallel\ E' \mid \mathcal{C}$ and $E, E' \vdash_{\mathsf{inf}} y : (V, \mathcal{M}) \ \parallel\ E'' \mid \mathcal{C}$.

$$\frac{E(y) = \mathcal{V} \quad \mathcal{V}' \text{ fresh} \quad E' = [\mathcal{V}''/\mathcal{V}]E \quad \mathcal{C} = \{\mathcal{V} \downarrow \mathcal{V}', \mathcal{V}' \le V, \mathcal{V}'' \le_{\mathcal{M}} \mathcal{V}'\}}{E \vdash_{\mathsf{inf}} y : (V, \mathcal{M}) \ \parallel\ E' \mid \mathcal{C}}$$

$$\frac{E(y) = \mathcal{V}_1 \quad \mathcal{V}'_1 \text{ fresh} \quad \mathcal{C} = \{\mathcal{V}_1 \downarrow \mathcal{V}'_1, \mathcal{V}'_1 \le V\} \quad E'(y) = \mathcal{V}_2 \quad \mathcal{V}'_2 \text{ fresh} \quad E'' = [\mathcal{V}''_2/\mathcal{V}_2]E' \quad \mathcal{C}' = \{\mathcal{V}_2 \downarrow \mathcal{V}'_2, \mathcal{V}'_2 \le V, \mathcal{V}''_2 \le_{\mathcal{M}} \mathcal{V}'_2\}}{E; E' \vdash_{\mathsf{inf}} y : (V, \mathcal{M}) \ \parallel\ E'' \mid \mathcal{C} \cup \mathcal{C}'}$$

We also add some definitions for the merge predicate.

$$\frac{\mathcal{V}'_1, \mathcal{V}'_2, V \text{ fresh} \quad \mathcal{V}_1 \ne \mathcal{V}_2 \quad \texttt{merge}(E'_1, E'_2, E, \mathcal{C}') \quad E'_1 = [\mathcal{V}'_1/\mathcal{V}_1, \mathcal{V}'_2/\mathcal{V}_2]E_1 \quad E'_2 = [\mathcal{V}'_1/\mathcal{V}_1, \mathcal{V}'_2/\mathcal{V}_2]E_2 \quad \mathcal{C} = \mathcal{C}' \cup \{\mathcal{V}_1 \downarrow \mathcal{V}'_1, \mathcal{V}_2 \downarrow \mathcal{V}'_2, \mathcal{V}'_1 \le V, \mathcal{V}'_2 \le V\}}{\texttt{merge}((y \mapsto \mathcal{V}_1, E_1), (y \mapsto \mathcal{V}_2, E_2), (y \mapsto (V, \mathcal{M}_\emptyset), E), \mathcal{C})}$$

$$\frac{\mathcal{V}', V' \text{ fresh} \quad \mathcal{C} = \mathcal{C}' \cup \{\mathcal{V} \downarrow \mathcal{V}', \mathcal{V}' \le V', (V, \mathcal{M}) \le V'\} \quad E'_1 = [\mathcal{V}'/\mathcal{V}]E_1 \quad E'_2 = [\mathcal{V}'/\mathcal{V}]E_2 \quad \texttt{merge}(E'_1, E'_2, E, \mathcal{C}')}{\texttt{merge}((y \mapsto \mathcal{V}, E_1), (y \mapsto (V, \mathcal{M}), E_2), (y \mapsto (V', \mathcal{M}_\emptyset), E), \mathcal{C})}$$

$$\frac{\mathcal{V} \text{ fresh} \quad E'_1 = [\mathcal{V}/\mathcal{V}_1, \mathcal{V}/\mathcal{V}_2]E_1 \quad E'_2 = [\mathcal{V}/\mathcal{V}_1, \mathcal{V}/\mathcal{V}_2]E_2 \quad \mathcal{C} = \mathcal{C}' \cup \{\mathcal{V}_1, \mathcal{V}_2 \downarrow \mathcal{V}\} \quad \texttt{merge}(E'_1, E'_2, E, \mathcal{C}')}{\texttt{merge}((\texttt{this} \mapsto \mathcal{V}_1, E_1), (\texttt{this} \mapsto \mathcal{V}_2, E_2), (\texttt{this} \mapsto \mathcal{V}, E), \mathcal{C})}$$

The inference rules generates some new types of constraints:

$$\mathcal{V} \downarrow \mathcal{V}' \quad \mathcal{V}_1, \mathcal{V}_2 \downarrow \mathcal{V}' \quad \mathcal{V} \leq V \quad \mathcal{V}_F = @[\,]$$
$$\mathcal{V} \leq_{(m,V)} \mathcal{V}' \quad \mathcal{V} \leq_{\mathcal{M}} \mathcal{V}' \quad \mathcal{V} \leq [m : (V, \psi)].$$

For the definition of constraint satisfiability, we define a few rules in addition to those in Section IV-B. If S is a satisfiable solution to the constraint set $\mathcal{C}$, then

$$\begin{aligned}
\mathcal{V} \leq_{(m,V)} \mathcal{V} \in \mathcal{C} &\Rightarrow S(\mathcal{V}) \leq_{(m,S(V))} S(\mathcal{V}') \\
\mathcal{V} \leq V \in \mathcal{C} &\Rightarrow S(\mathcal{V}) \leq S(V) \\
\mathcal{V} \leq_{\mathcal{M}} \mathcal{V}' \in \mathcal{C} &\Rightarrow S(\mathcal{V}) \leq_{S(\mathcal{M})} S(\mathcal{V}') \\
\mathcal{V} \downarrow \mathcal{V}' \in \mathcal{C} &\Rightarrow S(\mathcal{V}) \downarrow S(\mathcal{V}') \\
\mathcal{V}_1, \mathcal{V}_2 \downarrow \mathcal{V}' \in \mathcal{C} &\Rightarrow S(\mathcal{V}_1), S(\mathcal{V}_2) \downarrow S(\mathcal{V}') \\
\mathcal{V} \leq [m : (V, \psi)] \in \mathcal{C} &\Rightarrow S(\mathcal{V}) \leq [m : (S(V), \psi)] \\
\mathcal{V}_F = @[\,] \in \mathcal{C} &\Rightarrow S(\mathcal{V}_F) = @[\,]
\end{aligned}$$

where $\varsigma \leq [m : (u, \circ)]$ and $\varsigma \leq [m : (u, *)]$ are defined as below.

$$\frac{\varsigma(m) = (u, \psi) \quad \bullet \leq \psi}{\varsigma \leq [m : (u, \circ)]} \qquad \frac{\varsigma(m) = (u, \psi) \quad \psi \leq \bullet}{\varsigma \leq [m : (u, *)]}$$

We add some closure rules for the new forms of constraints.

Rule 16 to 18 propagates constraints related to extensions or updates to objects of singleton types.

$$\begin{array}{rl} \mathcal{V} \leq [m : (V, \psi)], \\ \mathcal{V}' \leq [m : (V', \psi')] \end{array} \longrightarrow V \leq V', V' \leq V \tag{15}$$

$$\mathcal{V} \leq_{(m,V)} \mathcal{V}' \longrightarrow \mathcal{V} \leq [m : (V, *)] \tag{16}$$

$$\begin{array}{rl} \mathcal{V} \leq_{(m,V)} \mathcal{V}', \\ \mathcal{V}' \leq [m' : (V', \psi)] \\ m' \neq m \ \vee \ \psi = \circ \end{array} \longrightarrow \mathcal{V} \leq [m' : (V', \psi)] \tag{17}$$

$$\begin{array}{rl} \mathcal{V} \leq_{(m,V)} \mathcal{V}', \\ \mathcal{V} \leq [m' : (V', \bullet)], \\ m' \neq m \end{array} \longrightarrow \mathcal{V}' \leq [m' : (V', \bullet)] \tag{18}$$

Rule 19 to 21 are for the interfacing between singleton types and obj-types.

$$\mathcal{V} \leq V, V \leq [m : (V', \psi)] \longrightarrow \begin{array}{l} \mathcal{V} \leq [m : (V', \psi)], \\ \mathcal{V} \leq [m : (V', \circ)] \end{array} \tag{19}$$

$$\mathcal{V} \downarrow \mathcal{V}', \mathcal{V} \leq [m : (V, \psi)] \longrightarrow \mathcal{V}' \leq [m : (V, \psi)] \tag{20}$$

$$\mathcal{V} \downarrow \mathcal{V}', \mathcal{V}' \leq [m : (V, \bullet)] \longrightarrow \mathcal{V} \leq [m : (V, \bullet)] \tag{21}$$

Rule 22 to 24 are for merging two singleton types into another singleton type.

$$\begin{array}{rl} \mathcal{V}_1, \mathcal{V}_2 \downarrow \mathcal{V}, \\ \mathcal{V}_1 \leq [m : (V_1, *)], \\ \mathcal{V}_2 \leq [m : (V_2, *)] \end{array} \longrightarrow \begin{array}{l} \mathcal{V} \leq [m : (V_1, *)], \\ \mathcal{V} \leq [m : (V_2, *)] \end{array} \tag{22}$$

$$\begin{array}{rl} \mathcal{V}_1, \mathcal{V}_2 \downarrow \mathcal{V}, \\ i \in \{1, 2\}.\ \mathcal{V}_i \leq [m : (V, \circ)] \end{array} \longrightarrow \mathcal{V} \leq [m : (V, \circ)] \tag{23}$$

$$\begin{array}{rl} \mathcal{V}_1, \mathcal{V}_2 \downarrow \mathcal{V}, \\ \mathcal{V} \leq [m : (V, \bullet)] \end{array} \longrightarrow \begin{array}{l} \mathcal{V}_1 \leq [m : (V, \bullet)], \\ \mathcal{V}_2 \leq [m : (V, \bullet)] \end{array} \tag{24}$$

The closure rules 25 and 26 are for the constraints of the form $\mathcal{V} \leq_{\mathcal{M}} \mathcal{V}'$, which are similar to those for $(V, \mathcal{M}) \leq V'$.

$$\begin{array}{rl} \mathcal{V} \leq_{\mathcal{M}} \mathcal{V}', \\ \mathcal{V}' \leq [m : (V, \psi)] \end{array} \longrightarrow \mathcal{V} \leq [m : (V, \psi)] \tag{25}$$

$$\begin{array}{rl} \mathcal{V} \leq_{\mathcal{M}} \mathcal{V}', \\ \mathcal{V} \leq [m : (V, \bullet)], \\ \mathcal{M} \subseteq \texttt{mSet}, m \notin \texttt{mSet} \end{array} \longrightarrow \mathcal{V}' \leq [m : (V, \bullet)] \tag{26}$$

The definition of consistency is similar to what we had before. In additional to the existing rules in Section IV-C, a constraint set $\mathcal{C}$ is inconsistent if

1) $\mathcal{V} \leq \texttt{bt} \in \mathcal{C}$
2) $\mathcal{V} \leq (V_0, \mathcal{M}_0) \times (V_1, \mathcal{M}_1) \to V_2 \in \mathcal{C}$
3) $V \leq [m : (\_, *)] \in \mathcal{C}$
4) $\{\mathcal{V} \leq [m : (V, \_)], \mathcal{V} = @[\,]\} \in \mathcal{C}$.

where the last rule replaces $V_F \leq [m : (V, \bullet)] \in \mathcal{C}$ in the previous consistency rules. A singleton type cannot be a base type or a function type, and an obj-type cannot have a member labeled with $*$ either. The initial type of the self pointer of a constructor function may not have any members.

We define the solution S for a constraint set $\mathcal{C}$ for its $\mathcal{V}$ variables so that $S(\mathcal{V}) = @[m_i : (u_i, \psi_i)^{i \in 1..n}]$ where $\forall i \in 1..n$, $u_i = S(V)$ for some V such that $[m_i : (V, \_)] \in \mathsf{Upper}_{\mathcal{C}}(\mathcal{V})$ and

1) $\psi_i = \bullet$ if $X = \{\bullet, \circ\}$, $\{*, \circ\}$, or $\{*, \bullet, \circ\}$,
2) $\psi_i = \circ$ if $X = \{\circ\}$, and
3) $\psi_i = *$ if $X = \{\bullet\}$, $\{*\}$, or $\{*, \bullet\}$.

where $X = \{\psi \mid [m_i : (\_, \psi)] \in \mathsf{Upper}_{\mathcal{C}}(\mathcal{V})\}$.

The label ψ_i of $S(\mathcal{V})$ is $*$ when $* \in X$. We also let ψ_i be $*$ if $X = \{*, \bullet\}$ since accessing the member m_i of $\mathcal{V}$ adds a constraint of the form $\mathcal{V} \leq [m_i : (V, \bullet)]$. We let ψ_i be $\bullet$ if $X = \{*, \bullet, \circ\}$ or $\{*, \circ\}$. The reason is that we keep track of whether a member of a singleton type $\mathcal{V}$ also exists in the type V where $\mathcal{V} \leq V$ by propagating constraints of the form $\mathcal{V} \leq [m_i : (V', \circ)]$ through Rule 19. When both constraints of the form $\mathcal{V} \leq [m_i : (V', *)]$ and $\mathcal{V} \leq [m_i : (V', \circ)]$ are present, it indicates that m_i has to be a definite member in $S(\mathcal{V})$.