Type Inference for Scripting languages 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.

1. Introduction

JavaScript is a widely used scripting languages for Web applications. 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 is a dynamic language, it cannot statically determine 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 members added to objects with some design variations. One design choice, taken by Anderson et al.'s type inference algorithm [4], 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 design choice, seen in Recency Types [10] and Bono and Fisher's calculus with object extensions [5], 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, one can also have features of both approaches above in one type system [7].

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 method/field add operation and self-inflicted extension [7], which is the extension that an object made to itself upon receiving a message. We do not model implicit extension of objects through function parameters though this type of extensions can be treated in a way similar to the extensions to self.

1.1 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
```

Figure 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, we create a `Form` object and call its `handle` method before and after calling its `set` method. 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 [4] 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.

```
function F(x) {
    this.a = 1;
    this.b = "one";
}
x1 = new F(0);
x2 = new F(0);
x1.b = true;
x1.c = 2;
x2.c = false;
```

Figure 2. Strong updates and unrestricted extensions to new objects

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 self-inflicted 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 2. Next, we formalize a type system on a small subset of JavaScript to support self-inflicted extension. We present the syntax, type rules, and operational semantics in Section 3. We explain the details of type inference algorithm and its correctness in Section 4. We explain the extension to add singleton types to new objects in Section 5.

2. Approach

We follow the design of Anderson's type system [4] 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. The inferred type of a function also includes a (possibly empty) set of names of members that are added to the receiver object during a call to the function.

Consider the example in Figure 1, the type of the variable `x` at line 14 can be written as

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

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_{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_{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. Therefore, the type of x at line 17 is

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

Hence, it is safe to call `handle` then.

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

$$(t_0, M) \times \tau_1 \to \tau_2$$

where t_0 is the type of `this` pointer, τ_1 and τ_2 are parameter and return type respectively. The meta variable τ ranges over object types, function types, and primitive types such as int, and a `top` type. M is a set of names of the members that are added to the receiver object by the function. The type of the function `setHandle` is then

$$t_{\mathtt{setter}} = (t_0, \{\mathtt{handle}\}) \times t_{\mathtt{handler}} \to \mathsf{int}$$

where t_0 and $t_{\mathtt{handle}}$ are defined as

$$t_0 = [\mathtt{handle} : (t_{\mathtt{handler}}, \circ)]$$

$$t_{\mathtt{handler}} = (t'_0, \emptyset) \times \mathsf{int} \to \mathsf{int}$$

$$t'_0 = [\,]\,.$$

The set M of a function type also includes members added by self-inflicted extensions within the function. That is, if a function f of type $(t_0, M) \times \tau_1 \to \tau_2$ calls function g of the type $(t'_0, M') \times \tau'_1 \to \tau_2$ on `this` pointer, then M must include M'.

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 ς_{x1} and ς_{x2} are the types of `x1` and `x2` after the last assignment.

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

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

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

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`.

```
function f(y) {
   y.a = 1;
   return y;
}
x = new F(0);
```

```
z = f(x);
x.b = true;
x.c = 2;
```

If the type of x starts with ς_{x}, then it becomes ς_{x}' after the call.

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

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

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

A singleton type may also have potential members as well. Suppose that we change the function `f` in the previous example so that it extends its parameter `y` with a `c` member.

```
function f(y) {
   y.c = 1;
   return y;
}
x = new F(0);
z = f(x);
x.c = 2;
```

The type of `x` before and after the call `f(x)` are:

$$\varsigma_{\mathsf{x}} = @[\mathsf{a} : (\mathsf{int}, *), \mathsf{b} : (\mathsf{string}, *)],$$
$$\varsigma_{\mathsf{x}}' = @[\mathsf{a} : (\mathsf{int}, \bullet), \mathsf{b} : (\mathsf{string}, *), \mathsf{c} : (\mathsf{int}, \circ)].$$

Notice that ς_{x}' now has a potential member `c` with `int` type. Any subsequent update to the `c` member of `x` has to be integers.

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

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

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

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

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

3. Formalization

In this section, we present a formalization of our type system. We explain the syntax, type rules, and the operational semantics, and prove the soundness of the type system. The details of type inference are covered in Section 4.

This formalization is for self-inflicted extension only. Additions to the type system are discussed in Section 5.

3.1 Syntax

We select a small subset of the JavaScript language that includes member select, member update/add, method calls, and object creation 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 [6].

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.

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

Figure 3. Syntax

3.2 Static semantics

We have four kinds of types: function type, object type, integer type, and a top type and the meta variable τ ranges over them.

τ	$::=$	t	name of function and object types
	$\mid$	top	super type of function and object types
	$\mid$	int	integer

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

t	$=$	$[m_i : (\tau_i, \psi_i)^{i\in i..n}]$	object type
t	$=$	$(t_0, M) \times \tau_1 \to \tau_2$	function type
ψ	$::=$	$\circ$	potential
	$\mid$	$\bullet$	definite

The meta variable ψ ranges over the label $\circ$ and $\bullet$, which indicates whether a member is potentially or definitely present. In a function type $(t_0, M) \times \tau_1 \to \tau_2$, t_0 is the type of `this`, which is always an object type while the parameter and return type τ_1 and τ_2 can be any types, and M represents a set of member names.

Subtyping The subtyping relation of types is defined as follows.

- Each object/function type is a subtype of top. Subtyping relation is reflexive and transitive.

$$t \leq \mathsf{top} \quad \tau \leq \tau \quad \frac{\tau \leq \tau' \quad \tau' \leq \tau''}{\tau \leq \tau''}$$

- 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.

$$\frac{t = (t_0, M) \times \tau_1 \rightarrow \tau_2 \quad t' = (t_0, M) \times \tau_1 \rightarrow \tau_2}{t \leq t'}$$

$$\frac{t = (t_0, M) \times \tau_1 \rightarrow \tau_2}{t \leq (t_0, M) \times \tau_1 \rightarrow \tau_2}$$

- 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.

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

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

  An object type t is a subtype of t' if t has a superset of members and the common members have the same types while their labels follow a partial order defined as $\psi \leq \psi$ and $\bullet \leq \circ$.

$$\frac{\forall m.\ t'(m) = (\tau, \psi') \Rightarrow (t(m) = (\tau, \psi)\ \wedge\ \psi \leq \psi')}{t \leq t'}$$

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

$$\frac{t(m) = (\tau, \psi) \quad \psi \leq \psi'}{t \leq [m : (\tau, \psi')]}$$

- We define another subtyping relationship $\leq_M$ to represent the member extensions of an object type so that $t \leq_M t'$ iff t is the same as t' except that each member in M must be definite in t.

$$\frac{\begin{array}{c}\forall m \notin M.\ t(m) = t'(m) \\ \forall m \in M.\ t(m) = (\tau, \bullet)\ \wedge\ t'(m) = (\tau, \psi)\end{array}}{t \leq_M t'}$$

3.2.1 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. Assume $\Gamma(n) = \mathsf{int}$ for any integer constant n. For any variable or name of function/constructor in the domain of Γ, we define

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

In Rule (T-Fn), we also use Γ to map a distinguished variable l to the set of members that are added to the receiver object during a method call.

A judgment of the form $\Gamma \vdash s \ \| \ \Gamma'$ asserts 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 mapping of function and constructor to 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, M is a set of member names. The set includes the members of the receiver object that are added or updated during a function call.

For a constructor function 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.

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

$$\frac{\begin{array}{c}\Gamma(f) \leq (t, M) \times \tau \rightarrow \tau' \\ \Gamma' = \Gamma[\mathtt{this} \mapsto t, x \mapsto \tau, \mathsf{l} \mapsto \emptyset] \quad \Gamma' \vdash s \ \| \ \Gamma'' \\ \Gamma''(z) \leq \tau' \quad M = \Gamma''(\mathsf{l})\end{array}}{\Gamma \vdash \mathsf{function}\ f(x)\{s; \mathsf{return}\ z\}} \qquad \text{T-Fn}$$

$$\frac{\begin{array}{c}\Gamma(F) = \tau \rightarrow t \\ \Gamma' = \Gamma[\mathtt{this} \mapsto t_0, x \mapsto \tau, \mathsf{l} \mapsto \emptyset] \quad \Gamma' \vdash s \ \| \ \Gamma'' \\ \Gamma''(\mathtt{this}) \leq t \quad \mathsf{def}(t_0) = \emptyset\end{array}}{\Gamma \vdash \mathsf{function}\ F(x)\{s\}} \qquad \text{T-Ctr}$$

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

Rule (T-Ctr) uses a helper function $\mathsf{def}(t)$ that returns the set of names of definite members in an object type t.

$$\mathsf{def}(t) = \{m \mid t(m) = (\tau, \bullet)\}$$

3.2.2 Type rules for statements

Type rules for statements are shown in Figure 5.

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 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 update the type of y so that its member m_j becomes definite (regardless of the original label of m_j) after the statement is executed.

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, i.e. with the label $\bullet$.

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. Also, if the called method adds a set of members denoted by the set M to `this`, then we update the type of the receiver object so that each method with name in M becomes definite in the type of the receiver.

For statements of the form $\mathtt{this}.m_j = z$ and $x = \mathtt{this}.m_j(z)$, Rule (T-Upd) and (T-Invk) also update the special variable l in the type environment Γ.

For example, consider the following program fragment:

```
this.m1 = 1;
this.m2 = 2;
x = this.setter(3);
```

where the method `setter` adds the members `m2` and `m3` to `this`. If before the execution of these statements, $\Gamma(\mathsf{l}) = \emptyset$, then after they are executed, $\Gamma(\mathsf{l}) = \{$ `m1, m2, m3` $\}$.

3.3 Operational semantics

We define a big-step semantics for our language in Figure 6. First, we give a few definitions used in the semantics.

$$\frac{x \notin \mathsf{dom}(\Gamma)}{\Gamma \vdash \mathsf{var}\ x \ \|\ \Gamma[x \mapsto \tau]} \quad \text{T-Dec}$$

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

$$\frac{\begin{array}{c}\Gamma(y) = t \quad t \leq [m_j : (\tau_j, \circ)] \quad \Gamma(z) \leq \tau_j \\ t' \leq_{\{m_j\}} t \quad \Gamma' = \Gamma[y \mapsto t'] \quad y \neq \mathtt{this} \Rightarrow \Gamma'' = \Gamma' \\ y = \mathtt{this} \Rightarrow \Gamma'' = \Gamma'[\mathsf{I} \mapsto \Gamma(\mathsf{I}) \cup \{m_j\}]\end{array}}{\Gamma \vdash y.m_j = z \ \|\ \Gamma''} \quad \text{T-Upd}$$

$$\frac{\Gamma(F) = \tau \to t \quad \Gamma(z) \leq \tau \quad \Gamma' = \Gamma[x \mapsto t]}{\Gamma \vdash x = \mathsf{new}\ F(z) \ \|\ \Gamma'} \quad \text{T-New}$$

$$\frac{\Gamma(y) \leq [m_j : (\tau_j, \bullet)] \quad \Gamma' = \Gamma[x \mapsto \tau_j]}{\Gamma \vdash x = y.m_j \ \|\ \Gamma'} \quad \text{T-Sel}$$

$$\frac{\begin{array}{c}\Gamma(y) \leq [m_j : (t_j, \bullet)] \quad t_j \leq (t_0, M) \times \tau_1 \to \tau_2 \\ \Gamma(z) \leq \tau_1 \quad \Gamma(y) \leq t_0 \quad t' \leq_M \Gamma(y) \\ \Gamma' = \Gamma[y \mapsto t', x \mapsto \tau_2] \quad y \neq \mathtt{this} \Rightarrow \Gamma'' = \Gamma' \\ y = \mathtt{this} \Rightarrow \Gamma'' = \Gamma'[\mathsf{I} \mapsto \Gamma(\mathsf{I}) \cup M]\end{array}}{\Gamma \vdash x = y.m_j(z) \ \|\ \Gamma''} \quad \text{T-Invk}$$

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

Figure 5. Type rules for statements

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, an integer, or null.

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

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 $j \in 1..n$.

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

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

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

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

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

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

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

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

Figure 6. 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.

3.3.1 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 12. 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 $\mathsf{nullPtrEx}$ as shown in Figure 13. We tolerate this type of error.

3.4 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 $\Sigma, \Gamma \vdash v : \tau$ asserts that the value v is well-typed with the type τ.

$$\Sigma, \Gamma \vdash n : \mathsf{int} \quad \Sigma, \Gamma \vdash \mathsf{null} : t \quad \frac{\Sigma(\iota) \leq t}{\Sigma, \Gamma \vdash \iota : t} \quad \frac{\Gamma(f) \leq t}{\Sigma, \Gamma \vdash f : t}$$

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.

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

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

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

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 $(t, M) \times \tau_1 \to \tau_2$, then M correctly identified the added (or updated) members of the receiver object. 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 3.1. *If* $\Sigma, \Gamma \vdash H, \chi$ *and* $\Gamma \vdash s \ \| \ \Gamma'$*, then* $H, \chi, s \not\rightsquigarrow \mathsf{error}$*, and if* $H, \chi, s \rightsquigarrow H', \chi'$*, then* $\exists\, \Sigma'$ *such that* $\Sigma', \Gamma' \vdash H', \chi'$.

Based on Lemma 3.1 (the proof is omitted), we can conclude that well-typed programs will not lead to errors caused by accessing undefined members.

Theorem 3.2 (Type Soundness). *If* $\Gamma \vdash P \ \| \ \Gamma'$*, then* $\emptyset, \emptyset, P \not\rightsquigarrow \mathsf{error}$ *and if* $\emptyset, \emptyset, P \rightsquigarrow H, \chi$*, then* $\exists\, \Sigma$ *such that* $\Sigma, \Gamma' \vdash H, \chi$.

4. 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 first three rules in Figure 7 generate constraints from programs, functions, and constructors. The judgment $E \vdash_{\mathsf{inf}} P \mid \mathcal{C}$ generates 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, the judgment $E \vdash_{\mathsf{inf}} Fn \mid \mathcal{C}$ generates a set of constraints $\mathcal{C}$ from a function or constructor declaration Fn based on the environment E.

Moreover, the variable $\mathcal{M}$ in Figure 7 corresponds to a set of member names and for each constructor function F, we create a unique type variable V_F for the initial type of `this` pointer.

$$\frac{E \vdash_{\mathsf{inf}} Fn_i \mid \mathcal{C}_i\ \forall i \in 1..n \quad E \vdash_{\mathsf{inf}} s \ \| \ E' \mid \mathcal{C}_0}{E \vdash_{\mathsf{inf}} Fn_i^{i\in 1..n}\ s \mid \bigcup_{i\in 0..n} \mathcal{C}_i}$$

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

$$\frac{\begin{array}{c} E' = E[x \mapsto V_{arg}, \mathtt{this} \mapsto V_F, \mathsf{l} \mapsto \emptyset] \quad E' \vdash_{\mathsf{inf}} s \ \| \ E'' \mid \mathcal{C}' \\ E(F) = V_{arg} \to V_{res} \quad \mathcal{C} = \mathcal{C}' \cup \{V_F \leq [\,], E''(\mathtt{this}) \leq V_{res}\} \end{array}}{E \vdash_{\mathsf{inf}} \mathsf{function}\ F(x)\{s\} \mid \mathcal{C}}$$

$$\frac{V\ \mathsf{fresh} \quad E' = E[x \mapsto V]}{E \vdash_{\mathsf{inf}} \mathsf{var}\ x \ \| \ E' \mid \emptyset}$$

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

$$\frac{\begin{array}{c} V_{y.m}, V_y, \mathcal{M}\ \mathsf{fresh} \quad E' = E[y \mapsto V_y] \quad y \neq \mathtt{this} \Rightarrow E'' = E' \\ y = \mathtt{this} \Rightarrow E'' = E'[\mathsf{l} \mapsto E(\mathsf{l}) \cup \mathcal{M}] \\ \mathcal{C} = \{E(y) \leq [m : (V_{y.m}, \circ)], E(z) \leq V_{y.m}, V_y \leq_{\mathcal{M}} E(y), m \in \mathcal{M}\} \end{array}}{E \vdash_{\mathsf{inf}} y.m = z \ \| \ E'' \mid \mathcal{C}}$$

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

$$\frac{\begin{array}{c} V_{y.m}, V_y, V_{\mathtt{this}}, V_{arg}, V_{res}, \mathcal{M}\ \mathsf{fresh} \quad E' = E[y \mapsto V_y, x \mapsto V_{res}] \\ \mathcal{C} = \{E(y) \leq [m : (V_{y.m}, \bullet)], V_{y.m} \leq (V_{\mathtt{this}}, \mathcal{M}) \times V_{arg} \to V_{res}, \\ E(z) \leq V_{arg}, V_y \leq_{\mathcal{M}} E(y), E(y) \leq V_{\mathtt{this}}\} \\ y \neq \mathtt{this} \Rightarrow E'' = E' \\ y = \mathtt{this} \Rightarrow E'' = E'[\mathsf{l} \mapsto E(\mathsf{l}) \cup \mathcal{M}] \end{array}}{E \vdash_{\mathsf{inf}} x = y.m(z) \ \| \ E'' \mid \mathcal{C}}$$

$$\frac{E(F) = V_{arg} \to V_{res} \quad \mathcal{C} = \{E(z) \leq V_{arg}\} \quad E' = E[x \mapsto V_{res}]}{E \vdash_{\mathsf{inf}} x = \mathsf{new}\ F(z) \ \| \ E' \mid \mathcal{C}}$$

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

Figure 7. Inference rules to generate constraints from a program

The inference rules make sure that each variable in the generated constraint set is unique.

The rest of the rules in Figure 7 generate type constraints from statements. Each judgment of the inference rule for statements is in the form of:

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

which generates a set of constraints $\mathcal{C}$ from a statement s with initial environment E and produces another environment E'. The environment E maps integers to `int` type, maps variables, function names to type variables, and constructor names to types of the form $V \to V'$, and it maps a special variable l to $\mathcal{N} = \emptyset \mid \bigcup_{i\in 1..n} \mathcal{M}_i$. l is added to the environment in the inference rule for functions and it is initially mapped to $\emptyset$ and may later be mapped to a union of set variables of the form $\emptyset \cup \mathcal{M}_1 \cup \ldots \cup \mathcal{M}_n$. To simplify notation, we write $\emptyset \cup \mathcal{M}_1 \cup \ldots \cup \mathcal{M}_n$ as $\mathcal{M}_1 \cup \ldots \cup \mathcal{M}_n$.

$$E(n) = \mathsf{int} \quad \frac{E = [\ldots F \mapsto V \to V' \ldots]}{E(F) = V \to V'}$$

$$\frac{E = [\ldots z \mapsto V \ldots]}{E(z) = V} \quad \frac{E = [\ldots \mathsf{l} \mapsto \mathcal{N} \ldots]}{E(\mathsf{l}) = \mathcal{N}}$$

After generating type constraints, we have a set of constraints of the following format:

$$\begin{array}{ll} \mathsf{int} \le V & V \le (V_0, \mathcal{M}) \times V_1 \to V_2 \\ V \le [\,] & V \le [m : (V', \psi)] \\ V \le V' & V \le_{\mathcal{M}} V' \\ m \in \mathcal{M} & \mathcal{M} = \mathcal{N} \end{array}$$

where $\mathcal{N} = \emptyset \mid \bigcup_{i \in 1..n} \mathcal{M}_i$.

4.1 Closure rules

Closure rules are shown in Figure 8, where the meta variables U and W are defined as follows:

$$\begin{array}{lll} U & ::= & \mathsf{int} \mid V \\ W & ::= & U \mid [\,] \mid [m : (V, \psi)] \mid (V, M) \times V_1 \to V_2 \end{array}$$

Rule 1 applies transitive closure to subtyping relations. Rule 2 and 3 ensure that int can only be subtype of itself. Rule 4 and 5 check constraints on object and function types with common lower bounds. Rule 6 propagates set membership for $\mathcal{M}$ variables. Note that in Rule 6, the constraint $\mathcal{M} = \bigcup_{i \in 1..n} \mathcal{M}_i$ can also represent constraint of the form $\mathcal{M} = \mathcal{M}_1$ when $n = 1$. Rule 7, 8, and 9 apply closure rules to member extension constraints. Rule 10 applies closure rules 1–9 to a constraint set to obtain a possibly larger constraint set. Rule 11 adds additional constraints to a constraint set that is closed with respect to Rule 10.

4.2 Constraint satisfiability

After applying closure rules, we obtain constraints of the form:

$$\begin{array}{ll} U \le W & V \le_{\mathcal{M}} V' \\ m \in \mathcal{M} & \mathcal{M} = \mathcal{N}. \end{array}$$

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

$$\begin{array}{rcl} S(\mathsf{int}) & = & \mathsf{int} \\ S([m : (V, \psi)]) & = & [m : (S(V), \psi)] \\ S((V_0, \mathcal{M}) \times V_1 \to V_2) & = & (S(V_0), S(\mathcal{M})) \times S(V_1) \to S(V_2) \\ S(\emptyset) = \emptyset & & S([\,]) = [\,] \\ S(\bigcup_{i \in 1..n} \mathcal{M}_i) & = & \bigcup_{i \in 1..n} S(\mathcal{M}_i) \end{array}$$

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

$$\begin{array}{rcl} U \le W \in \mathcal{C} & \Rightarrow & S(U) \le S(W) \\ V \le_{\mathcal{M}} V' \in \mathcal{C} & \Rightarrow & S(V) \le_{S(\mathcal{M})} S(V') \\ \{m \in \mathcal{M}\} \subseteq \mathcal{C} & \Rightarrow & m \in S(\mathcal{M}) \\ \mathcal{M} = \mathcal{N} \in \mathcal{C} & \Rightarrow & S(\mathcal{M}) = S(\mathcal{N}) \\ V_F \text{ appears in } \mathcal{C} & \Rightarrow & \mathsf{def}(S(V_F)) = \emptyset \end{array}$$

4.3 Constraint closure

Before solving a constraint set $\mathcal{C}$, we will compute its closure.

Definition A constraint set $\mathcal{C}$ is AClosed if $\mathcal{C} \longrightarrow_A \mathcal{C}$. Let $\mathsf{AClosure}(\mathcal{C}) = \mathcal{C}'$ if $\mathcal{C} \longrightarrow_A^* \mathcal{C}'$ and $\mathcal{C}' \longrightarrow_A \mathcal{C}'$, where $\longrightarrow_A$ is defined by Rule 10 and $\longrightarrow_A^*$ is a transitive closure of $\longrightarrow_A$.

Because the closure rules do not add any variables to the set, the closure of a constraint set has fixed number of variables and is bounded in size. Since $\longrightarrow_A$ is monotone and increasing, we can always find AClosure of a constraint set.

Definition A constraint set $\mathcal{C}$ is Closed if $\mathcal{C} \longrightarrow_B \mathcal{C}$. We define $\mathsf{Closure}(\mathcal{C}) = \mathcal{C}'$ if $\mathsf{AClosure}(\mathcal{C}) \longrightarrow_B^* \mathcal{C}'$ and $\mathcal{C}' \longrightarrow_B \mathcal{C}'$, where $\longrightarrow_B$ is defined by Rule 11 and $\longrightarrow_B^*$ is a transitive closure of $\longrightarrow_B$.

If $\mathcal{C} \longrightarrow_B \mathcal{C}'$, then $\mathcal{C}'$ remains AClosed because the constraints that may be added by Rule 11 are only different from those added by Rule 7 in the member labels. Since Rule 1–9 do not depend on labels, the new constraints added by Rule 11 cannot cause any of Rule 1–9 be applied again. Therefore, $\mathsf{Closure}(\mathcal{C})$ is also AClosed. Since a constraint set of finite variables is bounded in size and $\longrightarrow_B$ is monotone and increasing, we can always find the Closure of a constraint set $\mathcal{C}$.

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

1. $\{V \le \mathsf{int},\ V \le [\,]\} \subseteq \mathcal{C}$
2. $\{V \le \mathsf{int},\ V \le [m : (V, \psi)]\} \subseteq \mathcal{C}$
3. $\{V \le \mathsf{int},\ V \le (V_0, \mathcal{M}) \times V_1 \to V_2\} \subseteq \mathcal{C}$
4. $\{V \le [\,],\ V \le (V_0, \mathcal{M}) \times V_1 \to V_2\} \subseteq \mathcal{C}$
5. $\{V \le [m : (V, \psi)],\ V \le (V_0, \mathcal{M}) \times V_1 \to V_2\} \subseteq \mathcal{C}$
6. $\{m \in \mathcal{M},\ \mathcal{M} = \bigcup_{i \in 1..n} \mathcal{M}_i\} \subseteq \mathcal{C}$ and $\{m \in \mathcal{M}_i\} \not\subseteq \mathcal{C}$, $\forall i \in 1..n$,
7. $V_F \le [m : (V, \bullet)] \in \mathcal{C}$.

The first three rules of inconsistency make sure that a type variable cannot be both integer and an object type (or a function type) at the same time. The fourth and fifth rules make sure a type variable cannot be both an object type and a function type at the same time. The sixth rule makes sure that there is no conflict in solution to $\mathcal{M}$ variables. For example, if $\mathcal{C}$ is closed and $\{m \in \mathcal{M}', \mathcal{M} = \mathcal{M}', \mathcal{M} = \bigcup_{i \in 1..n} \mathcal{M}_i\} \subseteq \mathcal{C}$ then by Rule 6, $m \in \mathcal{M}$ has to be in $\mathcal{C}$ and any solution S to $\mathcal{C}$ will have $m \in S(\mathcal{M})$ but we also want $S(\mathcal{M}) = \bigcup_{i \in 1..n} S(\mathcal{M}_i)$. Thus, $\mathcal{C}$ is inconsistent if it does not contain $m \in \mathcal{M}_i$ for some i.

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 $V \le [m : (V', \bullet)]$. If the member is not defined before the access, then the closure rules will eventually generate $V_F \le [m : (V', \bullet)]$, where V_F represents the return type of the object's constructor, otherwise, only $V_F \le [m : (V', \circ)]$ will be generated.

4.5 Constraint solution

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

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

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

1. $S(\mathcal{M}) = \{m \mid (m \in \mathcal{M}) \in \mathcal{C}\}$;
2. $S(V) = \mathsf{int}$ if $\mathsf{int} \in \mathsf{Upper}_{\mathcal{C}}(V)$;

$$U \leq V, V \leq W \longrightarrow U \leq W \tag{1}$$
$$\mathsf{int} \leq V \longrightarrow V \leq \mathsf{int} \tag{2}$$
$$V \leq \mathsf{int} \longrightarrow \mathsf{int} \leq V \tag{3}$$
$$V \leq [m : (V', \psi')], V \leq [m : (V'', \psi'')] \longrightarrow V' \leq V'', V'' \leq V' \tag{4}$$
$$\begin{array}{c} V \leq (V_0, \mathcal{M}) \times V_1 \to V_2 \\ V \leq (V_0', \mathcal{M}') \times V_1' \to V_2' \end{array} \longrightarrow \begin{array}{l} V_0 \leq V_0', V_0' \leq V_0, V_1 \leq V_1', V_1' \leq V_1, \\ V_2 \leq V_2', V_2' \leq V_2, \mathcal{M} = \mathcal{M}', \mathcal{M}' = \mathcal{M} \end{array} \tag{5}$$
$$m \in \mathcal{M}_j,\ j \in \{1..n\},\ \mathcal{M} = \bigcup_{i \in 1..n} \mathcal{M}_i \longrightarrow m \in \mathcal{M} \tag{6}$$
$$V \leq_{\mathcal{M}} V', V \leq [m : (V'', \psi)] \longrightarrow V' \leq [m : (V'', \circ)] \tag{7}$$
$$V \leq_{\mathcal{M}} V', V' \leq [m : (V'', \psi)] \longrightarrow V \leq [m : (V'', \psi)] \tag{8}$$
$$V \leq_{\mathcal{M}} V', m \in \mathcal{M}, V' \leq [m : (V'', \circ)] \longrightarrow V \leq [m : (V'', \bullet)] \tag{9}$$
$$\frac{\forall i \in 1..k.\ c_i \in \mathcal{C} \quad c_1, .., c_k \longrightarrow c_1', .., c_n'}{\mathcal{C} \longrightarrow_A \mathcal{C} \cup \{c_1', .., c_n'\}} \tag{10}$$
$$\frac{\mathcal{C} \longrightarrow_A \mathcal{C} \quad \{V \leq_{\mathcal{M}} V', V \leq [m : (V_0, \bullet)], V'' \leq V'\} \subseteq \mathcal{C} \quad \{m \in \mathcal{M}\} \not\subseteq \mathcal{C}}{\mathcal{C} \longrightarrow_B \mathcal{C} \cup \{V' \leq [m : (V_0, \bullet)], V'' \leq [m : (V_0, \bullet)]\}} \tag{11}$$

Figure 8. Closure rules

3. $S(V) = \mathsf{top}$, if $\mathsf{Upper}_{\mathcal{C}}(V)$ is empty or only has 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 $(V_0, \mathcal{M}) \times V_1 \to V_2 \in \mathsf{Upper}_{\mathcal{C}}(V)$, then t is defined by $t = (S(V_0), S(\mathcal{M})) \times S(V_1) \to S(V_2)$;
   (b) if $[\,]$ or $[m : \_] \in \mathsf{Upper}_{\mathcal{C}}(V)$, then $t = [m_i : (\tau_i, \psi_i)^{i \in N}]$, where $\forall i \in N$, $\tau_i = S(V')$ 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 `int` type and for other variables that cannot be object or function types, we set the variables to the type `top`. 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.

4.6 Type inference as constraint closure consistency

In this section, 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 4.5.

The proof for this section is omitted.

Lemma 4.1. *If $E \vdash_{\mathsf{inf}} P : \mathcal{C}'$ and $\mathcal{C} = \mathsf{Closure}(\mathcal{C}')$ is consistent, then $\mathcal{C}$ is satisfiable*

We can prove Lemma 4.1 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 $\mathcal{C}$.

Lemma 4.2. *If $\mathcal{C}$ is satisfiable, then $\mathsf{Closure}(\mathcal{C})$ is consistent.*

To prove Lemma 4.2, 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 4.3. *Given a program P where $E \vdash_{\mathsf{inf}} P \mid \mathcal{C}$, P is typable iff $\mathsf{Closure}(\mathcal{C})$ is consistent.*

From Theorem 4.3, we can conclude that our type inference algorithm is sound and complete with respect to our type system.

5. Allow strong updates to new objects

Since we assume that constructor functions always return new objects, we can assign singleton types to the return values of constructors. 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

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

where τ_i cannot be singleton types and ψ_i may be either $*$, $\bullet$, or $\circ$. As mentioned before, the label $*$ annotates members that can have strong updates. An object of singleton type can have unrestricted extensions as well. Also, given ς defined as above, $\varsigma(m_i) = (\tau_i, \psi_i)$ for all $i \in 1..n$ and $\varsigma(m_i) = \mathsf{undef}$ otherwise.

5.1 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 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.

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

where we define $* \leq \bullet$ and $* \leq *$.

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.

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

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 is passed to a parameter or assigned to a field.

This is illustrated in the following example where function `f` updates or extends the members `a` and `c` of its parameter `y`.

```
1  function F(d) {
2    this.a = 1;
3    this.b = "one";
4  }
5  function f(y) {
6     y.a = 1;
7     y.c = 2;
8     return y;
9  }
10 x = new F(0);
11 x1 = x;
12 z = f(x);
13 w = z.a;
```

The type of `x` on line 10 is ς_{x} and it becomes ς'_{x} on line 12.

$$\begin{aligned}\varsigma_{\mathsf{x}} &= @[\mathsf{a} : (\mathsf{int}, *), \mathsf{b} : (\mathsf{string}, *)]\\ \varsigma'_{\mathsf{x}} &= @[\mathsf{a} : (\mathsf{int}, \bullet), \mathsf{b} : (\mathsf{string}, *), \mathsf{c} : (\mathsf{int}, \circ)]\\ t_{\mathsf{y}} &= [\mathsf{a} : (\mathsf{int}, \circ), \mathsf{c} : (\mathsf{int}, \circ)]\end{aligned}$$

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 `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}. Because $\varsigma_{\mathsf{x}} \downarrow \varsigma'_{\mathsf{x}}$, we can safely change the type of x to ς'_{x} since ς'_{x} has all the members of ς_{x} with some of its $*$ members labeled as definite. 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 `z`'s type $t_{\mathsf{z}} = [\mathsf{a} : (\mathsf{int}, \bullet)]$. Therefore, the variable `x` can have strong updates even after it is assigned to variables of obj-types.

Since we need to downgrade singleton types in several places including update statements, object allocations, and method calls, we define a rule of the form $\Gamma \vdash x : \tau \ \ \| \ \ \Gamma'$. In particular, if the type of `x` is a subtype of τ, then the environment Γ remains the same. Otherwise, we downgrade the type of `x` from ς_{x} to ς'_{x} so that ς_{x} is a subtype of τ, and we replace all occurrences of ς_{x} in Γ with ς'_{x}, which is written as $[\varsigma'_{\mathsf{x}}/\varsigma_{\mathsf{x}}]\Gamma$. To see why this last step is necessary, consider the previous example where the variable `x1` and `x` are aliases of each other and they both have the type ς_{x}. If the type of `x1` remains ς_{x} after the call `f(x)`, then the update `x1.a = true` will change the member `z.a` to boolean, since `x1` and `z` refer to the same object. This would be inconsistent with the type of `z`.

$$\frac{\Gamma(z) = \varsigma \quad \varsigma \downarrow \varsigma' \quad \varsigma' \leq \tau \quad \Gamma' = [\varsigma'/\varsigma]\Gamma}{\Gamma \vdash z : \tau \ \| \ \Gamma'} \qquad \frac{\Gamma(z) \leq \tau}{\Gamma \vdash z : \tau \ \| \ \Gamma}$$

The substitution of singleton types in Γ is defined as:

$$\frac{\Gamma = \Gamma' \cup \{y \mapsto \varsigma\} \quad \Gamma'' = [\varsigma'/\varsigma]\Gamma'}{[\varsigma'/\varsigma]\Gamma = \Gamma'' \cup \{y \mapsto \varsigma'\}} \qquad \frac{\varsigma \text{ does not appear in } \Gamma}{[\varsigma'/\varsigma]\Gamma = \Gamma}$$

$$\frac{\Gamma = \Gamma' \cup \{y \mapsto \tau\} \quad \Gamma'' = [\varsigma'/\varsigma]\Gamma' \quad \tau \neq \varsigma}{[\varsigma'/\varsigma]\Gamma = \Gamma'' \cup \{y \mapsto \tau\}}$$

We also define a relation of the form $\varsigma' \leq_{(m,\tau)} \varsigma$ such that if an object of singleton type ς is assigned a value of type τ to its member m, then the resulting type of the object is ς'. If m is not defined in ς or it is labeled with $*$ in ς, then the object can receive strong update.

$$\frac{\begin{array}{c}\varsigma = @[m_i : (\tau_i, \psi_i)^{i \in 1..n}] \quad m_i \neq m, \forall i \in 1..n\\ \varsigma' = @[m_i : (\tau_i, \psi_i)^{i \in 1..n}, m : (\tau, *)]\end{array}}{\varsigma' \leq_{(m,\tau)} \varsigma}$$

$$\frac{\begin{array}{c}\varsigma = @[m_i : (\tau'_i, \psi'_i)^{i \in 1..n}] \quad \varsigma' = @[m_i : (\tau_i, \psi_i)^{i \in 1..n}]\\ \tau_j = \tau \quad \forall i \neq j, \tau_i = \tau'_i, \psi_i = \psi'_i\\ j \in 1..n \quad (\psi_j = \psi'_j = *) \ \vee \ (\tau_j = \tau'_j \ \wedge \ \psi'_j = \bullet \ \wedge \ \psi_j \neq *)\end{array}}{\varsigma' \leq_{(m_j,\tau)} \varsigma}$$

For example, the types of `this` at line 2 $\varsigma_{\mathtt{this}}$ and at line 3 $\varsigma'_{\mathtt{this}}$ have the relation $\varsigma'_{\mathtt{this}} \leq_{(\mathsf{b},\mathsf{string})} \varsigma_{\mathtt{this}}$.

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

Moreover, we define some relations for singleton types similar to those for obj-types.

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

$$\frac{\begin{array}{c}\forall m \notin M. \, \varsigma'(m) = \varsigma(m)\\ \forall m \in M. \, \varsigma'(m) = (\tau, \bullet), \ \varsigma(m) = (\tau, \psi), \ \bullet \leq \psi\end{array}}{\varsigma' \leq_M \varsigma}$$

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

$$\frac{\begin{array}{c}\Gamma' = \Gamma[\mathtt{this} \mapsto \varsigma, x \mapsto \tau_{arg}] \quad \Gamma' \vdash s \ \| \ \Gamma''\\ \Gamma(F) = \tau_{arg} \to \varsigma_{res} \quad \Gamma''(\mathtt{this}) \downarrow \varsigma_{res} \quad \mathsf{def}(\varsigma) = \emptyset\end{array}}{\Gamma \vdash \mathsf{function}\ F(x)\{s\}} \quad \text{T-Ctr}$$

where $\mathsf{def}(\varsigma) = \{m \mid \varsigma(m) = (\tau, \_)\}$.

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, new statements, and method calls as in Figure 9. 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 is labeled with $*$ or m_j is not defined in ς.

5.2 Type inference

We need to modify the type inference rules for constructor function, new statement, method call, and update in a way similar to the type rules as in Figure 10. We use the variable $\mathcal{V}$ to represent singleton types while V still represents obj-types.

The inference rules generates some new types of constraints:

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

For the definition of constraint satisfiability, we define a few rules in addition to those in Section 4.2. If S is a satisfiable solution

$$\dfrac{\begin{array}{lll} & & \Gamma \vdash z : \tau \;\parallel\; \Gamma' \\ \Gamma'(y) = \varsigma_y & \Rightarrow & \Gamma'' = [\varsigma'_y/\varsigma_y]\Gamma' \quad \varsigma'_y \leq_{(m,\tau)} \varsigma_y \\ \Gamma'(y) = t_y & \Rightarrow & t_y \leq [m : (\tau, \circ)] \quad t'_y \leq_{\{m\}} t_y \\ & & y \neq \mathtt{this} \Rightarrow \Gamma'' = \Gamma'[y \mapsto t'_y] \\ & & y = \mathtt{this} \Rightarrow \\ & & \Gamma'' = \Gamma'[y \mapsto t'_y, \mathsf{l} \mapsto \Gamma(\mathsf{l}) \cup \{m\}] \end{array}}{\Gamma \vdash y.m = z \;\parallel\; \Gamma''} \quad \text{T-Upd}$$

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

$$\dfrac{\begin{array}{lll} & & \Gamma(y) \leq [m : (t_{y.m}, \bullet)] \quad \Gamma \vdash z : \tau_{arg} \;\parallel\; \Gamma' \\ & & t_{y.m} \leq (t_{\mathtt{this}}, M) \times \tau_{arg} \to \tau_{res} \\ \Gamma'(y) = \varsigma_y & \Rightarrow & \Gamma' \vdash y : t_{\mathtt{this}} \;\parallel\; \Gamma_1 \quad \varsigma = \Gamma_1(y) \\ & & \varsigma'_y \leq_M \varsigma \quad \Gamma'' = [\varsigma'_y/\varsigma]\Gamma_1 \\ \Gamma'(y) = t_y & \Rightarrow & t'_y \leq_M t_y \quad t_y \leq t_{\mathtt{this}} \\ & & y \neq \mathtt{this} \Rightarrow \Gamma'' = \Gamma'[y \mapsto t'_y] \\ & & y = \mathtt{this} \Rightarrow \\ & & \Gamma'' = \Gamma'[y \mapsto t'_y, \mathsf{l} \mapsto \Gamma(\mathsf{l}) \cup M] \end{array}}{\Gamma \vdash x = y.m(z) \;\parallel\; \Gamma''[x \mapsto \tau_{res}]} \quad \text{T-Invk}$$

Figure 9. New type rules for statements

to the constraint set $\mathcal{C}$, then

$$\begin{array}{rcl} \mathcal{V} \leq_{(m,V)} \mathcal{V} \in \mathcal{C} & \Rightarrow & S(\mathcal{V}) \leq_{(m,S(V))} S(\mathcal{V}') \\ \mathcal{V} \leq V \in \mathcal{C} & \Rightarrow & S(\mathcal{V}) \leq S(V) \\ \mathcal{V} \leq_{\mathcal{M}} \mathcal{V}' \in \mathcal{C} & \Rightarrow & S(\mathcal{V}) \leq_{S(\mathcal{M})} S(\mathcal{V}') \\ \mathcal{V} \downarrow \mathcal{V}' \in \mathcal{C} & \Rightarrow & S(\mathcal{V}) \downarrow S(\mathcal{V}') \\ \mathcal{V} \leq [m : (V, \psi)] \in \mathcal{C} & \Rightarrow & S(\mathcal{V}) \leq [m : (S(V), \psi)] \\ \mathcal{V}_F \text{ appears in } \mathcal{C} & \Rightarrow & \mathsf{def}(S(\mathcal{V}_F)) = \emptyset. \end{array}$$

We add some closure rules for the new forms of constraints in Figure 11. The closure rules 17, 18, and 19 are for the constraints of the form $\mathcal{V} \leq_{\mathcal{M}} \mathcal{V}'$, which are similar to those for $V \leq_{\mathcal{M}} V'$. Rule 13, 14, and 20 propagates constraints related to extensions or updates to objects of singleton types. Rule 15, 16, and 21 are for the interfacing between singleton types and obj-types. Rule 19, 20, and 21 are applied to $\mathsf{AClosed}$ constraint set and the resulting constraint set is still $\mathsf{AClosed}$.

The definition of consistency is similar to what we had before. In additional to the existing rules in Section 4.4, a constraint set $\mathcal{C}$ is inconsistent if

1. $\mathcal{V} \leq \mathsf{int} \in \mathcal{C}$
2. $\mathcal{V} \leq (V_0, \mathcal{M}) \times V_1 \to V_2 \in \mathcal{C}$
3. $V \leq [m : (\_, *)] \in \mathcal{C}$
4. $\mathcal{V}_F \leq [m : (V, \_)] \in \mathcal{C}$.

where the last rule replaces $V_F \leq [m : (V, \bullet)] \in \mathcal{C}$ in the previous consistency rules. A singleton type cannot be an integer or 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 $S(\mathcal{V}) = [m_i : (\tau_i, \psi_i)^{i \in 1..n}]$ where $\forall i \in 1..n$, $\tau_i = S(V)$ for some V such that $[m_i : (V, \_)] \in \mathsf{Upper}_{\mathcal{C}}(\mathcal{V})$ and

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

$$\dfrac{\begin{array}{c} E' = E[x \mapsto V_{arg}, \mathtt{this} \mapsto \mathcal{V}_F] \\ E' \vdash_{\mathsf{inf}} s \;\parallel\; E'' \mid \mathcal{C} \quad E(F) = V_{arg} \to \mathcal{V}_{res} \end{array}}{E \vdash_{\mathsf{inf}} \mathsf{function}\ F(x)\{s\} \mid \mathcal{C} \cup \{E''(\mathtt{this}) \downarrow \mathcal{V}_{res}\}}$$

$$\dfrac{\begin{array}{lll} & & V \text{ fresh} \quad E \vdash_{\mathsf{inf}} z : V \;\parallel\; E' \mid \mathcal{C} \\ E'(y) = \mathcal{V}_y & \Rightarrow & \mathcal{V}'_y \text{ fresh} \quad E'' = [\mathcal{V}'_y/\mathcal{V}_y]E' \\ & & \mathcal{C}' = \{\mathcal{V}'_y \leq_{(m,V)} \mathcal{V}_y\} \\ E'(y) = V_y & \Rightarrow & V'_y, \mathcal{M} \text{ fresh} \\ & & \mathcal{C}' = \{V_y \leq [m : (V, \circ)], V'_y \leq_{\mathcal{M}} V_y, m \in \mathcal{M}\} \\ & & y \neq \mathtt{this} \Rightarrow E'' = E'[y \mapsto V'_y] \\ & & y = \mathtt{this} \Rightarrow \\ & & E'' = E'[y \mapsto V'_y, \mathsf{l} \mapsto E(\mathsf{l}) \cup \mathcal{M}] \end{array}}{E \vdash_{\mathsf{inf}} y.m = z \;\parallel\; E'' \mid \mathcal{C} \cup \mathcal{C}'}$$

$$\dfrac{\begin{array}{lll} & & V_{y.m}, V_{\mathtt{this}}, V_{arg}, V_{res}, \mathcal{M} \text{ fresh} \quad E \vdash_{\mathsf{inf}} z : V_{arg} \;\parallel\; E' \mid \mathcal{C}' \\ & & \mathcal{C} = \{E(y) \leq [m : (V_{y.m}, \bullet)], V_{y.m} \leq (V_{\mathtt{this}}, \mathcal{M}) \times V_{arg} \to V_{res}\} \\ E'(y) = \mathcal{V}_y & \Rightarrow & \mathcal{V}'_y \text{ fresh} \quad E' \vdash y : V_{\mathtt{this}} \;\parallel\; E_1 \mid \mathcal{C}_2 \\ & & \mathcal{V} = E_1(y) \quad E'' = [\mathcal{V}'_y/\mathcal{V}]E_1 \\ & & \mathcal{C}'' = \mathcal{C}_2 \cup \{\mathcal{V}'_y \leq_M \mathcal{V}\} \\ E'(y) = V_y & \Rightarrow & V'_y \text{ fresh} \quad \mathcal{C}'' = \{V'_y \leq_{\mathcal{M}} V_y, V_y \leq V_{\mathtt{this}}\} \\ & & y \neq \mathtt{this} \Rightarrow E'' = E'[y \mapsto V'_y] \\ & & y = \mathtt{this} \Rightarrow \\ & & E'' = E'[y \mapsto V'_y, \mathsf{l} \mapsto E(\mathsf{l}) \cup \mathcal{M}] \end{array}}{E \vdash_{\mathsf{inf}} x = y.m(z) \;\parallel\; E''[x \mapsto V_{res}] \mid \mathcal{C} \cup \mathcal{C}' \cup \mathcal{C}''}$$

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

$$\dfrac{\begin{array}{lll} E(z) = V & \Rightarrow & \mathcal{C} = \{E(z) \leq V\} \quad E' = E \\ E(z) = \mathcal{V} & \Rightarrow & \mathcal{V}' \text{ fresh} \quad E' = [\mathcal{V}'/\mathcal{V}]E \\ & & \mathcal{C} = \{\mathcal{V} \downarrow \mathcal{V}', \mathcal{V}' \leq V\} \end{array}}{E \vdash_{\mathsf{inf}} z : V \;\parallel\; E' \mid \mathcal{C}}$$

Figure 10. New type inference rules

3. $\psi_i = \bullet$ otherwise,

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 a member select statement adds a constraint of the form $\mathcal{V} \leq [m_i : (V, \bullet)]$. Reading a member of a singleton type does not make the member definite. Finally, we let ψ_i be $\bullet$ if $\{*, \circ\} \subseteq X$, or $\{\bullet, \circ\} \subseteq X$, or $X = \{\bullet\}$. The reason for this 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 15. 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 $S(\mathcal{V})$.

6. Related work

Our work is similar to the type inference system of Anderson et al. [4] 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 explicit extension where an object may extend itself through method calls on the object. We also

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

$$\mathcal{V} \leq_{(m,V)} \mathcal{V}' \quad \longrightarrow \quad \mathcal{V} \leq [m : (V, *)] \tag{13}$$

$$\mathcal{V} \leq_{(m,V)} \mathcal{V}', \quad \mathcal{V}' \leq [m' : (V', \psi)] \quad \longrightarrow \quad \mathcal{V} \leq [m' : (V', \psi)] \quad \text{where } m' \neq m \text{ or } \psi = \circ \tag{14}$$

$$\mathcal{V} \leq V, \quad V \leq [m : (V', \psi)] \quad \longrightarrow \quad \mathcal{V} \leq [m : (V', \psi)], \quad \mathcal{V} \leq [m : (V', \circ)] \tag{15}$$

$$\mathcal{V} \downarrow \mathcal{V}', \quad \mathcal{V} \leq [m : (V, \psi)] \quad \longrightarrow \quad \mathcal{V}' \leq [m : (V, \psi)] \tag{16}$$

$$\mathcal{V} \leq_{\mathcal{M}} \mathcal{V}', \mathcal{V}' \leq [m : (V, \psi)] \quad \longrightarrow \quad \mathcal{V} \leq [m : (V, \psi)] \tag{17}$$

$$\mathcal{V} \leq_{\mathcal{M}} \mathcal{V}', m \in \mathcal{M}, \mathcal{V}' \leq [m : (V, \circ)] \quad \longrightarrow \quad \mathcal{V} \leq [m : (V, \bullet)] \tag{18}$$

$$\frac{\mathcal{C} \longrightarrow_A \mathcal{C} \quad \{\mathcal{V} \leq_{\mathcal{M}} \mathcal{V}', \quad \mathcal{V} \leq [m : (V, \bullet)]\} \subseteq \mathcal{C} \quad \{m \in \mathcal{M}\} \not\subseteq \mathcal{C}}{\mathcal{C} \longrightarrow_B \mathcal{C} \cup \{\mathcal{V}' \leq [m : (V, \bullet)]\}} \tag{19}$$

$$\frac{\mathcal{C} \longrightarrow_A \mathcal{C} \quad \{\mathcal{V} \leq_{(m,V)} \mathcal{V}', \quad \mathcal{V} \leq [m' : (V', \bullet)]\} \subseteq \mathcal{C} \quad m' \neq m}{\mathcal{C} \longrightarrow_B \mathcal{C} \cup \{\mathcal{V}' \leq [m' : (V', \bullet)]\}} \tag{20}$$

$$\frac{\mathcal{C} \longrightarrow_A \mathcal{C} \quad \{\mathcal{V} \downarrow \mathcal{V}', \quad \mathcal{V}' \leq [m : (V, \bullet)]\} \subseteq \mathcal{C}}{\mathcal{C} \longrightarrow_B \mathcal{C} \cup \{\mathcal{V} \leq [m : (V, \bullet)]\}} \tag{21}$$

Figure 11. Additional closure rules

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. [7] 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.

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

Recency types of Heidegger and Thiemann [10, 11] 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. [12] have implemented a practical analyzer to detect possible runtime errors of JavaScript program. Their approach is based on abstract interpretation and uses recency information. The analyzer can report the absence of errors based on some inputs but it does not infer types.

Earlier work of Thiemann [22] proposed a type system for a subset of JavaScript language to detect conversion errors of JavaScript values. The type system models automatic conversions in JavaScript but it does not model recursive or flow sensitive types.

There are a number of studies on type inference for class-based languages. Palsberg et al. [13, 15] have developed a type inference algorithm based on ideas of flow analysis for object-oriented programs with inheritance, assignments, and late binding. The purpose is to guarantee all messages are understood while allowing polymorphic methods. The algorithm handles late binding with conditional constraints and solves the constraints by least fixed-point derivation. Similar algorithm was applied to object-based language SELF [3] that features objects with dynamic inheritance. Plevyak and Chien [18] extended this flow-based approach for better precision via an incremental algorithm. Further enhancement on precision and efficiency were made by Agesen in his Cartesian Product Algorithm [2], which was applied to type inference for Python programs to improve compiled code [20]. Eifrig et al. [8] developed a polymorphic, constraint-based type inference algorithm for a class-based language with polymorphic *recursively constrained* types. The goal was to mitigate the tradeoff between inheritance and subtyping. The recursively constrained types are also used in a type inference algorithm for Java [23] to verify the correctness of downcasts. Their inference algorithm extends Agesen's Cartesian Product Algorithm with the ability to analyze data polymorphic programs.

DRuby [9] 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 [14] with recursive types and subtyping for Abadi Cardelli object calculus [1], which has method override and subsumption but not object extension. similar algorithms were developed for inferring object types for an object calculus with covariant read-only fields [17] and supporting record concatenation [16].

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

7. Conclusion and discussion

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 [19]. However, our system is lack of many important features found in real world JavaScript programs such as prototypes, variadic functions, eval function, member deletion, and objects as associative arrays. Also, our type system does not allow depth subtyping on object types or support parametric polymorphism.

Some improvement seems possible with the current design. Currently, the type of a function argument is not extended after the call returns. For example, if we pass a variable `x` to a function `addSize`, which extends the parameter with an additional member `size`, the variable `x` has the same type before and after the call: $[\texttt{size} : (\mathsf{int}, \circ), \ldots]$, with the potential `size` member.

```
x = new Form();
addSize(x);
```

It seems straightforward to have this type of extension so that after the call returns, the type of `x` becomes $[\texttt{size} : (\mathsf{int}, \bullet), \ldots]$. The problem is to only identify members added to the function parameter before it is overwritten by other value. Also, we would like to include branch statements and prototypes in the formalization. A new object of singleton type can have strong updates until it becomes a function's prototype.

References

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

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

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

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

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

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

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

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

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

[10] P. Heidegger and P. Thiemann. Recency Types for Dynamically-Typed, Object-Based Languages. In *International Workshop on Foundations of Object-Oriented Languages (FOOL'09)*, 2009.

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

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

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

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

[15] J. Palsberg and M. I. Schwartzbach. Object-Oriented Type Inference. *SIGPLAN Not.*, 26(11):146–161, 1991.

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

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

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

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

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

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

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

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

A. Error propagation

Figure 12 contains the rules for propagating runtime errors caused by accessing an undefined object member or calling an undefined function or constructor. Figure 13 describes the conditions that lead to null pointer exceptions.

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

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

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

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

Figure 12. Error of accessing undefined members or functions

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

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

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

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

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

Figure 13. Null pointer exception

B. 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.

```
function Form(a) {
   this.set = setter;
}
function setter(b) {
   this.handle = b;
   return 0;
}
function handler(c) {
   return 0;
}
// main
x = new Form(1);
y = x.set(handler);
z = x.handle(1);
```

We first show the constraints generated from each function in Figure 14, 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. Also, we have three related type variables: V_x and V_{x_1} are for the types of `x` before and after the call `x.set(handler)` while V_{x_2} is for the type of `x` after the call `x.handle(1)`.

Functions	Generated constraints	
Form	$V_{Form} \leq [\mathtt{set} : (V_1, \circ)]$ $V_2 \leq_{\mathcal{M}_1} V_{Form} \quad \mathtt{set} \in \mathcal{M}_1$	$V_{Form} \leq [\,]$ $V_{setter} \leq V_1$
setter	$V_{setter} \leq (V_3, \mathcal{M}_2) \times V_b \to V_4$ $V_3 \leq [\mathtt{handle} : (V_5, \circ)] \quad V_b \leq V_5$ $V_6 \leq_{\mathcal{M}_3} V_3 \quad \mathtt{handle} \in \mathcal{M}_3$	$\mathsf{int} \leq V_4$ $V_3 \leq [\,]$ $\mathcal{M}_2 = \mathcal{M}_3$
handler	$V_{handler} \leq (V_7, \mathcal{M}_4) \times V_c \to V_8$ $\mathsf{int} \leq V_8$	 $V_7 \leq [\,]$
main	$V_2 \leq V_x$ $V_x \leq [\mathtt{set} : (V_9, \bullet)]$ $V_9 \leq (V_{10}, \mathcal{M}_5) \times V_{11} \to V_{12}$ $V_{handler} \leq V_{11}$ $V_{x_1} \leq [\mathtt{handle} : (V_{13}, \bullet)]$ $V_{13} \leq (V_{14}, \mathcal{M}_6) \times V_{15} \to V_{16}$ $\mathsf{int} \leq V_{15}$	$\mathsf{int} \leq V_a$ $V_{12} \leq V_y$ $V_x \leq V_{10}$ $V_{x_1} \leq_{\mathcal{M}_5} V_x$ $V_{16} \leq V_z$ $V_{x_1} \leq V_{14}$ $V_{x_2} \leq_{\mathcal{M}_6} V_{x_1}$

Figure 14. 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 15. We can verify that the closure of the original constraints set is consistent. In particular, the upper bound of V_{Form} is $\{[\,], [\mathtt{set} : (V_1, \circ)]\}$ and satisfies the consistency rule that it may not contain object types with definite members.

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

$$\mathcal{M}_2 = \mathcal{M}_3 \quad \mathcal{M}_2 = \mathcal{M}_5 \quad \mathcal{M}_5 = \mathcal{M}_2 \quad \mathcal{M}_4 = \mathcal{M}_6 \quad \mathcal{M}_6 = \mathcal{M}_4$$
$$\mathtt{set} \in \mathcal{M}_1 \quad \mathtt{handle} \in \mathcal{M}_3 \quad \mathtt{handle} \in \mathcal{M}_2 \quad \mathtt{handle} \in \mathcal{M}_5$$

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 16.

C. Proof of type soundness

Some of the proof omitted from the paper is included here.

Lemma C.1. *If* $\Gamma \vdash s \ \| \ \Gamma'$, $\forall m \in \Gamma(\mathsf{l})$, $\Gamma(\mathtt{this}) \leq [m : (\_, \bullet)]$, *then* $\Gamma'(\mathtt{this}) \leq_{\Gamma'(\mathsf{l})} \Gamma(\mathtt{this})$.

Proof. If $\Gamma' = \Gamma$, then by definition of $\leq_M$, $\Gamma(\mathtt{this}) \leq_{\Gamma(\mathsf{l})} \Gamma(\mathtt{this})$ since every m in $\Gamma(\mathsf{l})$ is a definite member in $\Gamma(\mathtt{this})$.

Type variables	Type upper bound
V_a	int
V_1	$V_9, (V_{10}, \mathcal{M}_5) \times V_{11} \to V_{12}$
V_2	$[\texttt{set} : (V_1, \bullet)], V_x, [\texttt{set} : (V_9, \bullet)], V_{10}$
	$V_3, [\texttt{handle} : (V_5, \circ)], [\,], [\texttt{handle} : (V_{13}, \circ)]$
V_{Form}	$[\,], [\texttt{set} : (V_1, \circ)]$
V_{setter}	$(V_3, \mathcal{M}_2) \times V_b \to V_4, V_1, V_9$
	$(V_{10}, \mathcal{M}_5) \times V_{11} \to V_{12}$
V_3	$[\texttt{handle} : (V_5, \circ)], [\,], V_{10}$
V_4	int, V_{12}, V_y
V_b	$V_5, V_{11}, V_{13}, (V_{14}, \mathcal{M}_6) \times V_{15} \to V_{16}$
V_5	$V_{13}, (V_{14}, \mathcal{M}_6) \times V_{15} \to V_{16}$
$V_{handler}$	$(V_7, \mathcal{M}_4) \times V_c \to V_8, V_{11}, V_b, V_5$
	$(V_{14}, \mathcal{M}_6) \times V_{15} \to V_{16}, V_{13}$
V_c	V_{15}, int
V_7	$[\,], V_{14}$
V_8	int, V_{16}
V_x	$[\texttt{set} : (V_9, \bullet)], V_{10}, V_3, [\texttt{handle} : (V_5, \circ)], [\,]$
	$[\texttt{handle} : (V_{13}, \circ)]$
V_9	$(V_{10}, \mathcal{M}_5) \times V_{11} \to V_{12}, V_1$
V_{10}	$V_3, [\texttt{handle} : (V_5, \circ)], [\,]$
V_{11}	$V_b, V_5, V_{13}, (V_{14}, \mathcal{M}_6) \times V_{15} \to V_{16}$
V_{12}	V_y, V_4, int
V_{x_1}	$[\texttt{set} : (V_9, \bullet)], [\texttt{handle} : (V_{13}, \bullet)], V_{14}$
	$[\texttt{handle} : (V_5, \circ)]$
V_{13}	$(V_{14}, \mathcal{M}_6) \times V_{15} \to V_{16}, V_5$
V_{14}	$V_7, [\,]$
V_{15}	int, V_c
V_{16}	V_z, int, V_8
V_y	int
V_z	int

Figure 15. Type upper bounds of each type variable

Γ' may be different from Γ only if s is a member update/add or a method call on `this` variable, or s is a sequence statements.

If s is $\texttt{this}.m_j = z$, then $\Gamma'(\mathsf{l}) = \Gamma(\mathsf{l}) \cup \{m_j\}$, $\Gamma(\texttt{this}) \leq [m_j : (\_, \circ)]$, and $\Gamma'(\texttt{this}) \leq_{\{m_j\}} \Gamma(\texttt{this})$. Thus, $\Gamma'(\texttt{this}) \leq [m_j : (\_, \bullet)]$ and $\Gamma'(\texttt{this}) \leq_{\Gamma'(\mathsf{l})} \Gamma(\texttt{this})$.

If s is $x = \texttt{this}.m(z)$, $\Gamma(\texttt{this}) \leq [m : (t, \bullet)]$, and $t \leq (t_0, M) \times \tau_1 \to \tau_2$, then $\Gamma'(\mathsf{l}) = \Gamma(\mathsf{l}) \cup M$, $\Gamma'(\texttt{this}) \leq_M \Gamma(\texttt{this})$. By the definition of $\leq_M$, $\forall m \in M$, $\Gamma(\texttt{this}) \leq [m : (\_, \circ)]$ and $\Gamma'(\texttt{this}) \leq [m : (\_, \bullet)]$. Therefore, $\Gamma'(\texttt{this}) \leq_{\Gamma'(\mathsf{l})} \Gamma(\texttt{this})$.

If $s = S_1; S_2$ and $\Gamma \vdash s_1 \;\|\; \Gamma', \Gamma' \vdash s_2; \Gamma''$, by induction, $\Gamma'(\texttt{this}) \leq_{\Gamma'(\mathsf{l})} \Gamma(\texttt{this})$, which means $\forall m \in \Gamma'(\mathsf{l})$, $\Gamma'(\texttt{this}) \leq [m : (\_, \bullet)]$. By the induction again, we have $\Gamma''(\texttt{this}) \leq_{\Gamma''(\mathsf{l})} \Gamma'(\texttt{this})$. Since $\Gamma'(\mathsf{l}) \subseteq \Gamma''(\mathsf{l})$, by definition of $\leq_M$, we have $\Gamma''(\texttt{this}) \leq_{\Gamma''(\mathsf{l})} \Gamma(\texttt{this})$.

□

Lemma C.2. *If* $\Gamma \vdash \mathsf{function}\ f(x)\{s; \mathsf{return}\ z\}$, $\Gamma(f) = (t, M) \times \tau_1 \to \tau_2$, $\Gamma' = \Gamma[\texttt{this} \mapsto t, x \mapsto \tau_1, \mathsf{l} \mapsto \emptyset]$, $\Gamma' \vdash s \;\|\; \Gamma''$, *and* $M = \Gamma''(\mathsf{l})$, *then* $\Gamma''(\texttt{this}) \leq_M t$.

Names	Corresponding types
a	int
	$t_1 = (t_3, \{\texttt{handle}\}) \times t_b \to$ int
	$t_2 = [\texttt{set} : (t_1, \bullet),\ \texttt{handle} : (t_5, \circ)]$
Form	int $\to t_x$
	$t_5 = (t_7, \emptyset) \times$ int $\to$ int
	$t_3 = [\texttt{handle} : (t_5, \circ)]$
b	$t_b = (t_7, \emptyset) \times$ int $\to$ int
setter	$t_{setter} = (t_3, \{\texttt{handle}\}) \times t_b \to$ int
	$t_7 = [\,]$
c	int
handler	$t_{handler} = (t_7, \emptyset) \times$ int $\to$ int
x	$t_x = [\texttt{set} : (t_1, \bullet),\ \texttt{handle} : (t_5, \circ)]$
x_1	$t_{x_1} = [\texttt{set} : (t_1, \bullet),\ \texttt{handle} : (t_5, \bullet)]$
y	int
z	int

Figure 16. Constraint solution

Proof. Since $\Gamma'(\mathsf{l}) = \emptyset$ and $\Gamma' \vdash s \;\|\; \Gamma''$, from Lemma C.1, $\Gamma''(\texttt{this}) \leq_{\Gamma''(\mathsf{l})} \Gamma'(\texttt{this})$, which is $\Gamma''(\texttt{this}) \leq_M t$. □

Lemma C.3. *If* $\Sigma, \Gamma \vdash H, \chi$, $\chi(y) = \iota$, *and* $\Gamma(y)(m) = (\tau, \bullet)$, *then* $H(\iota)(m) = v$ *where* $\Sigma, \Gamma \vdash v : \tau$.

Lemma 3.1 *If* $\Sigma, \Gamma \vdash H, \chi$ *and* $\Gamma \vdash s \;\|\; \Gamma'$, *then* $H, \chi, s \not\rightsquigarrow$ error, *and if* $H, \chi, s \rightsquigarrow H', \chi'$, *then* $\exists\, \Sigma'$ *such that* $\Sigma', \Gamma' \vdash H', \chi'$.

Proof. We prove by induction analysis on reduction rules applied. For the proof, we need an additional invariant – $\Sigma'(\iota) \leq \Sigma(\iota)$ for any $\iota \in \mathsf{dom}(\Sigma)$.

R-Dec Let s be var x. Then $H, \chi, s \rightsquigarrow H, \chi[x \mapsto$ null$]$. Since null can have any type and if $\Gamma \vdash s \;\|\; \Gamma[x \mapsto \tau]$, then $\Gamma[x \mapsto \tau] \vdash H, \chi[x \mapsto$ null$]$.

R-Asn Let s be $x = z$. By Rule (T-Asn), $\Gamma' = \Gamma[x \mapsto \Gamma(z)]$. By the induction hypothesis, we have $\Sigma, \Gamma \vdash \chi(z) : \Gamma(z)$. Hence, $\Sigma, \Gamma' \vdash H, \chi[x \mapsto \chi(z)]$ holds.

R-Sel Let s be $y.m_j$. By Rule (T-Sel), $\Gamma(y) \leq [m_j : (\tau_j, \bullet)]$ and $\Gamma' = \Gamma[x \mapsto \tau_j]$. Then $\exists\iota$ such that $\chi(y) = \iota$. From $\Gamma \vdash H, \chi$ and Lemma C.3, $\exists v_j$ such that $H(\iota)(m_j) = v_j$ and $\Sigma, \Gamma \vdash v_j : \tau_j$. Thus, $H, \chi, x = y.m_j \rightsquigarrow H, \chi[x \mapsto v_j]$. From $\Sigma, \Gamma \vdash v_j : \tau_j$, we have $\Sigma, \Gamma' \vdash H, \chi[x \mapsto v_j]$.

R-Upd Let s be $y.m_j = z$. Let $\chi(y) = \iota$. Then $H, \chi, y.m_j = z \rightsquigarrow H', \chi$, where $H' = H[\iota \mapsto H(\iota)[m_j \mapsto \chi(z)]]$. By Rule (T-Upd), $\Gamma(z) \leq \tau_j$ where $\Gamma(y) \leq [m_j : (\tau_j, \circ)]$. From $\Sigma, \Gamma \vdash H, \chi$, we have $\Sigma, \Gamma \vdash \chi(z) : \Gamma(z)$. Also by Rule (T-Upd), $\Gamma' = \Gamma[y \mapsto t]$, and $t \leq_{\{m_j\}} \Gamma(y)$. Let $\Sigma' = \Sigma[\iota \mapsto t']$ and $t' \leq_{\{m_j\}} \Sigma(\iota)$. From $\Sigma(\iota) \leq \Gamma(y)$, we have $t' \leq t$. Then $\Sigma', \Gamma' \vdash H', \chi$. The only difference between Σ' and Σ is the type of ι and $\Sigma'(\iota) \leq \Sigma(\iota)$.

R-Invk Let s be $x = y.m_j(z)$. By Rule (T-Invk), $\Gamma(y) \leq [m_j : (t_j, \bullet)]$ and $t_j \leq (t_0, M) \times \tau_1 \to \tau_2$. From $\Gamma \vdash H, \chi$ and Lemma C.3, $\exists f$ such that $H(\chi(y))(m_j) = f$ and $\Gamma(f) \leq t_j$. Let $\mathsf{lookup}(f, \chi(\mathsf{ft})) = \mathsf{function}\ f(x')\{s'; \mathsf{return}\ z'\}$. Let $\Gamma_0 = \Gamma_{\texttt{init}}[x' \mapsto \tau_1, \texttt{this} \mapsto t_0]$. Then by Rule (T-Fn), $\Gamma_0 \vdash s' \;\|\; \Gamma'_0$ and $\Gamma'_0(\texttt{this}) \leq_M t_0$.
By Rule (T-Invk), $\Gamma(y) \leq t_0$ and $\Gamma(z) \leq \tau_1$. Thus, $\Sigma, \Gamma_0 \vdash H, \chi'$ where $\chi' = \{x' \mapsto \Gamma(z), \texttt{this} \mapsto \Gamma(y), \mathsf{ft} \mapsto \chi(\mathsf{ft})\}$. Thus, by induction hypothesis, $\exists\Sigma'$ such that if $H, \chi', s' \rightsquigarrow$

H', χ'', then $\Sigma', \Gamma'_0 \vdash H', \chi''$ and $\Gamma'_0(z') \leq \tau_2$. Let $\chi(y) = \iota$, and $\Gamma' = \Gamma[y \mapsto t, x \mapsto \tau_2]$, where $t \leq_M \Gamma(y)$. By Lemma C.2, $\Gamma'_0(\texttt{this}) \leq_M t_0$. Since $\Sigma'(\iota) \leq \Gamma'_0(\texttt{this})$, each $m \in M$ is definite member of $\Sigma'(\iota)$. From $\Gamma'(y) \leq_M \Gamma(y)$, the only differences between $\Gamma(y)$ and $\Gamma'(y)$ are the labels on members in M. Also since $\Sigma'(\iota) \leq \Sigma(\iota) \leq \Gamma(y)$ it follows that $\Sigma'(\iota) \leq \Gamma'(y)$.

By the induction hypothesis, $\forall \iota \in \mathsf{dom}(\Sigma)$, $\Sigma'(\iota) \leq \Sigma(\iota)$. Thus, $\Sigma', \Gamma' \vdash \chi(y) : \Gamma'(y), \forall y \in \mathsf{dom}(\chi)$.

From Rule (T-Invk), we have $\Gamma'(x) = \tau_2$. From $\Sigma', \Gamma'_0 \vdash H', \chi''$, we have $\Sigma', \Gamma'_0 \vdash \chi''(z') : \Gamma'_0(z')$. From $\Gamma'_0(z') \leq \tau_2$, it follows that $\Sigma', \Gamma' \vdash H', \chi[x \mapsto \chi''(z')]$.

R-New Let s be $x = \mathsf{new}\ F(z)$ and $\mathsf{function}\ F(x')\{s'\}$ be the definition of F. From Rule (T-Ctr), $\exists\ t_0$ and τ such that $\Gamma_1 = \Gamma_{\texttt{init}}[\texttt{this} \mapsto t_0, x' \mapsto \tau]$, $\Gamma_1 \vdash s \ \|\ \Gamma_2, \Gamma_2(\texttt{this}) \leq t'_0$, and $\Gamma(F) = \tau \to t'_0$. By Rule (T-New), $\Gamma' = \Gamma[x \mapsto t'_0]$ and $\Gamma(z) \leq \tau$. Let ι be a new object label, $\chi_1 = \{\texttt{this} \mapsto \iota, x' \mapsto \chi(z)\}$, and $H_1 = H[\iota \mapsto []]$. Since $\mathsf{def}(t_0) = \emptyset$, it is clear that $\Sigma[\iota \mapsto t_0], \Gamma_1 \vdash H_1, \chi_1$. By the induction hypothesis, there exists Σ', Γ_2 such that if $H_1, \chi_1, s \rightsquigarrow H_2, \chi_2$, then $\Sigma', \Gamma_2 \vdash H_2, \chi_2$.

By the induction hypothesis, $\forall \iota \in \mathsf{dom}(\Sigma)$, $\Sigma'(\iota) \leq \Sigma(\iota)$. Thus, $\forall y \in \mathsf{dom}(\chi)$, $\Sigma', \Gamma' \vdash \chi(y) : \Gamma'(y)$.

Since $\Sigma'(\iota) \leq \Gamma_2(\texttt{this}) \leq t'_0 = \Gamma'(x)$, we have $\Sigma', \Gamma' \vdash H_2, \chi[x \mapsto \iota]$.

R-Seq The proof is by induction.

□

D. Proof of type inference as constraint closure consistency

Lemma D.1. *If $E \vdash_{\mathsf{inf}} P \mid \mathcal{C}'$ and $\mathcal{C} = \mathsf{Closure}(\mathcal{C}')$ is consistent, then $\{V \leq_{\mathcal{M}} V', m \in \mathcal{M}\} \subseteq \mathcal{C}$ implies $V' \leq [m : \_]$ is in $\mathcal{C}$.*

Proof. We prove by induction that if $\{V \leq_{\mathcal{M}} V', m \in \mathcal{M}\} \subseteq \mathcal{C}$ or $V \leq (V', \mathcal{M}) \times V_{arg} \to V_{res}$ is in $\mathcal{C}$, then $V' \leq [m : \_]$ is in $\mathcal{C}$.

Let's first consider how constraints with $\mathcal{M}$ variables are added to $\mathcal{C}'$ by the inference rules. For each member update, we generate constraints of the form $V \leq_{\mathcal{M}} V'$, $m \in \mathcal{M}$, and $V \leq [m : \_]$. For each method call, we generate constraints of the form $V_{y.m} \leq (V_{\texttt{this}}, \mathcal{M}) \times V_{arg} \to V_{res}$, $V_y \leq V_{\texttt{this}}$, and $V'_y \leq_{\mathcal{M}} V_y$. For each function, we generate constraints of the form $V_f \leq (V_{\texttt{this}}, \mathcal{M}) \times V_{arg} \to V_{res}$, $\mathcal{M} = \bigcup_{i \in 1..n} \mathcal{M}_i$, and for the function body, we generate constraints of the form $V_n \leq_{\mathcal{M}_n} \dots V_1 \leq_{\mathcal{M}_1} V_{\texttt{this}}$.

For the constraints generated from a method call, if $m \in \mathcal{M}$ is in $\mathcal{C}$, then it has to be the result of applying closure rule 5, where $\{V_{y.m} \leq (V'_{\texttt{this}}, \mathcal{M}') \times V'_{arg} \to V'_{res}, \mathcal{M}' = \mathcal{M}, m \in \mathcal{M}', V_{\texttt{this}} \leq V'_{\texttt{this}}\} \subseteq \mathcal{C}$. By the induction hypothesis, $V'_{\texttt{this}} \leq [m : \_]$ is in $\mathcal{C}$. Thus, $V_{\texttt{this}} \leq [m : \_]$ is in $\mathcal{C}$ as well. Also, since $V_y \leq V_{\texttt{this}}$ is in $\mathcal{C}$, $V_y \leq [m : \_]$ is in $\mathcal{C}$.

For the constraints generated from a function, if $m \in \mathcal{M}$ is in $\mathcal{C}$, then by consistency rule 6, there exists $\mathcal{M}_i$ so that $m \in \mathcal{M}_i$ is in $\mathcal{C}$. By the induction hypothesis, $V_i \leq [m : \_]$ is in $\mathcal{C}$. By closure rule 7 and 8, we have $V_{\texttt{this}} \leq [m : \_]$ in $\mathcal{C}$. □

Lemma 4.1 *If $E \vdash_{\mathsf{inf}} P : \mathcal{C}'$ and $\mathcal{C} = \mathsf{Closure}(\mathcal{C}')$ is consistent, then $\mathcal{C}$ is satisfiable.*

Proof. Since $\mathcal{C}$ is Closed, it is AClosed as well.

We show that $S = \mathsf{Solution}(\mathcal{C})$ satisfies each constraint in $\mathcal{C}$.

First, $S(\mathcal{M}) = \{m \mid m \in \mathcal{M}\}$. S apparently satisfies constraint of the form $m \in \mathcal{M}$. For constraint of the form $\mathcal{M} = \bigcup_{i \in 1..n} \mathcal{M}_i$, since $\mathcal{C}$ is AClosed, by Rule 6, if $(m \in \mathcal{M}_i) \in \mathcal{C}$ and $i \in 1..n$, then $m \in \mathcal{M}$ as well. Thus, $S(\mathcal{M}_i) \subseteq S(\mathcal{M})$. Since $\mathcal{C}$ is consistent, for each $(m \in \mathcal{M}) \in \mathcal{C}$, $\exists \mathcal{M}_i$ such that $(m \in \mathcal{M}_i) \in \mathcal{C}$. Therefore, $S(\mathcal{M}) = \bigcup_{i \in 1..n} S(\mathcal{M}_i)$. For constraint of the form $\mathcal{M} = \mathcal{M}'$ and $\mathcal{M}' = \mathcal{M}$, $(m \in \mathcal{M}) \in \mathcal{C}$ iff $(m \in \mathcal{M}') \in \mathcal{C}$. Thus, $S(\mathcal{M}) = S(\mathcal{M}')$.

Also, for each V_F appearing in $\mathcal{C}$, from the definition of consistency, $\mathsf{Upper}_{\mathcal{C}}(V_F)$ does not contain type of the form $[m : (V, \bullet)]$. Thus, by the definition of Solution, no member of $S(V_F)$ is definite, which means $\mathsf{def}(S(V_F)) = \emptyset$.

Next we consider constraints on types.

1. If $\mathsf{int} \leq V$ or $V \leq \mathsf{int} \in \mathcal{C}$, then $S(V) = \mathsf{int}$.
2. If $V \leq [m : (V', \psi)] \in \mathcal{C}$, then by the definition of consistency, $\mathsf{Upper}_{\mathcal{C}}(V)$ does not contain int or function types, Thus, $\exists t$ such that $S(V) = t$ and $t(m) = (S(V_1), \psi')$, where $[m : (V_1, \psi_1)] \in \mathsf{Upper}_{\mathcal{C}}(V)$, $\psi' = \bullet$ if $[m : (V_2, \bullet)] \in \mathsf{Upper}_{\mathcal{C}}(V)$ for some V_2 and $\psi' = \circ$ otherwise. By Rule 4, $V' \leq V_1$ and $V_1 \leq V' \in \mathcal{C}$, which implies $S(V') = S(V_1)$. Thus, $S(V) \leq S([m : (V', \psi)])$.
3. If $V \leq (V_0, \mathcal{M}) \times V_1 \to V_2 \in \mathcal{C}$, then by the definition of consistency, $\mathsf{Upper}_{\mathcal{C}}$ does not contain int or object types. Thus, $\exists t$, such that $S(V) = t$ and $t = (S(V'_0), S(\mathcal{M}')) \times S(V'_1) \to S(V'_2)$, where $(V'_0, \mathcal{M}') \times V'_1 \to V'_2 \in \mathsf{Upper}_{\mathcal{C}}(V)$. By Rule 5, $V_0 \leq V'_0, V'_0 \leq V_0, \mathcal{M} = \mathcal{M}', \mathcal{M}' = \mathcal{M}, V_1 \leq V'_1, V'_1 \leq V_1, V_2 \leq V'_2, V'_2 \leq V_2 \in \mathcal{C}$. Thus, $S(V_0) = S(V'_0)$, $S(\mathcal{M}) = S(\mathcal{M}')$, $S(V_1) = S(V'_1)$, $S(V_2) = S(V'_2)$, which means that $S(V) \leq S((V_0, \mathcal{M}) \times V_1 \to V_2)$.
4. If $V \leq V' \in \mathcal{C}$, we consider the following subcases:
   (a) $S(V') = \mathsf{int}$ iff $S(V) = \mathsf{int}$.
   (b) If $S(V') = \mathsf{top}$, then $S(V)$ is equal to some t. Thus, $S(V) \leq S(V')$.
   (c) If $V' \leq (V_0, \mathcal{M}) \times V_1 \to V_2 \in \mathcal{C}$, then $V \leq (V_0, \mathcal{M}) \times V_1 \to V_2 \in \mathcal{C}$ by Rule 1. Since $\mathcal{C}$ is consistent, $\mathsf{Upper}_{\mathcal{C}}(V) \supseteq \mathsf{Upper}_{\mathcal{C}}(V')$ and they do not contain int or object types. By the reasoning similar to case 3, $S(V) = t$ and $S(V') = t'$ for some t and t', where $\exists t_0, M, t_1, t_2$, such that $t = (t_0, M) \times t_1 \to t_2$ and $t' = (t_0, M) \times t_1 \to t_2$. Thus, $S(V) \leq S(V')$.
   (d) If $V' \leq [m : (V_1, \psi)]$, then by Rule 1, $V \leq [m : (V_1, \psi)]$. Since $\mathcal{C}$ is consistent, $\mathsf{Upper}_{\mathcal{C}}(V) \supseteq \mathsf{Upper}_{\mathcal{C}}(V')$ and they do not contain int or function types. Also, by the reasoning similar to case 2, $S(V) = t$ and $S(V') = t'$ for some t and t', where $t(m) = (t_1, \psi')$ and $t'(m) = (t_1, \psi'')$ for some t_1. Also, $\psi' \leq \psi''$ since if $\psi'' = \bullet$, then by the definition of Solution, we must assign $\bullet$ to ψ' as well. Thus, $S(V) \leq S(V')$.
5. $V \leq_{\mathcal{M}} V'$.
   Since $\mathcal{C}$ is closed, from Rule 7 and 8, $[m : (V'', \psi)] \in \mathsf{Upper}_{\mathcal{C}}(V)$ iff $\exists [m : (V'', \psi')] \in \mathsf{Upper}_{\mathcal{C}}(V')$, where $\psi \leq \psi'$. Also, from Rule 11, if $[m : (V'', \bullet)] \in \mathsf{Upper}_{\mathcal{C}}(V)$, $\mathcal{C} \longrightarrow_A \mathcal{C}$, and $m \in \mathcal{M}$ is not in $\mathcal{C}$ then $[m : (V'', \bullet)] \in \mathsf{Upper}_{\mathcal{C}}(V')$. Since $\mathcal{C}$ is AClosed and $m \in \mathcal{M}$ is not in $\mathcal{C}$, $m \notin S(\mathcal{M})$. Thus, for $m \notin S(\mathcal{M})$, $S(V)(m) = S(V')(m)$. Note that if m is not a member in $S(V)$, then $S(V)(m) = \mathsf{undef}$.
   Moreover, by Rule 9, if $m \in \mathcal{M}$ and $V' \leq [m : (V'', \circ)]$ are in $\mathcal{C}$, then $V \leq [m : (V'', \bullet)]$ is in $\mathcal{C}$. By Lemma D.1, if $m \in \mathcal{M}$ and $V \leq_{\mathcal{M}} V'$ are in $\mathcal{C}$, then $V' \leq [m : \_]$ is in $\mathcal{C}$. By Rule 7 and 8, there exists $V' \leq [m : (V'', \circ)]$ in $\mathcal{C}$. Therefore, if $m \in S(\mathcal{M})$, then $S(V)(m) = (S(V''), \bullet)$ and $S(V')(m) = (S(V''), \psi)$.
   Thus, by the definition of $\leq_M$, we have $S(V) \leq_{S(\mathcal{M})} S(V')$.

□

Lemma 4.2 *If $\mathcal{C}$ is satisfiable, then $\mathsf{Closure}(\mathcal{C})$ is consistent.*

Proof. We show that if S is a solution to $\mathcal{C}$ and $\mathcal{C} \longrightarrow_A \mathcal{C}'$ or $\mathcal{C} \longrightarrow_B \mathcal{C}'$, then S is a solution to $\mathcal{C}'$ as well. We perform a case analysis based on the closure rules used. Since Rule 10 uses Rule 1–9, we do not analyze it as a separate case.

Rule 1 In this case, if $U \leq V, V \leq W \in \mathcal{C}$, then $U \leq W \in \mathcal{C}'$. By the definition of subtyping relation, it is transitive. Therefore, if $S(U) \leq S(V)$ and $S(V) \leq S(W)$, then $S(U) \leq S(W)$.

Rule 2 If $\mathsf{int} \leq V \in \mathcal{C}$, then $\mathsf{int} \leq S(V)$ and $S(V) = \mathsf{int} \leq \mathsf{int}$.

Rule 3 If $V \leq \mathsf{int} \in \mathcal{C}$, then $S(V) \leq \mathsf{int}$ and $\mathsf{int} \leq S(V) = \mathsf{int}$.

Rule 4 If $V \leq [m : (V', \psi')], V \leq [m : (V'', \psi'')] \in \mathcal{C}$, then $V' \leq V'', V'' \leq V' \in \mathcal{C}'$. Since $S(V) \leq [m : (S(V'), \psi')]$ and $S(V) \leq [m : (S(V''), \psi')]$, by subtyping rules on object types, if $S(V) = (\tau, \psi)$, then $\tau = S(V') = S(V'')$. Thus, $S(V') \leq S(V'')$ and $S(V'') \leq S(V')$.

Rule 5 If $V \leq (V_0, \mathcal{M}) \times V_1 \to V_2, V \leq (V_0', \mathcal{M}') \times V_1' \to V_2' \in \mathcal{C}$ then $V_0 \leq V_0', V_0' \leq V_0, V_1 \leq V_1', V_1' \leq V_1, V_2 \leq V_2', V_2' \leq V_2, \mathcal{M} = \mathcal{M}', \mathcal{M}' = \mathcal{M} \in \mathcal{C}'$. By subtyping rules on function types, if $S(V) = (t, M) \times \tau_1 \to \tau_2$, then $S(\mathcal{M}) = S(\mathcal{M}') = M$, $t = S(V_0) = S(V_0')$, $\tau_1 = S(V_1) = S(V_1')$, and $\tau_2 = S(V_2) = S(V_2')$. Thus, S solves $\mathcal{C}'$.

Rule 6 If $m \in \mathcal{M}_j, j \in \{1..n\}, \mathcal{M} = \bigcup_{i \in 1..n} \mathcal{M}_i \in \mathcal{C}$, then $(m \in \mathcal{M}) \in \mathcal{C}'$. Since S is a solution to $\mathcal{C}$, we have $m \in S(\mathcal{M}_j)$ and $S(\mathcal{M}) = \bigcup_{i \in 1..n} S(\mathcal{M}_i)$. Thus, $m \in S(\mathcal{M})$.

Rule 7 If $V \leq_{\mathcal{M}} V', V \leq [m : (V'', \psi)] \in \mathcal{C}$, then $V' \leq [m : (V'', \circ)] \in \mathcal{C}'$. Since $S(V) \leq_S (\mathcal{M})V'$ and $S(V) \leq [m : (S(V'', \psi)]$, by definition of $\leq_M$, if $m \notin S(\mathcal{M})$, then $S(V)(m) = S(V')(m) = (\tau, \psi')$ for some τ and ψ', which means $S(V'') = \tau$. If $m \in S(\mathcal{M})$, then $S(V')(m) = (\tau, \circ)$ and $S(V)(m) = (\tau, \bullet)$. In both cases, $S(V') \leq [m : (S(V''), \circ)]$.

Rule 8 If $V \leq_{\mathcal{M}} V', V' \leq [m : (V'', \psi)] \in \mathcal{C}$, then $V \leq [m : (V'', \psi)] \in \mathcal{C}'$. If $m \notin S(\mathcal{M})$, then $S(V)(m) = S(V')(m)$, and together with $S(V') \leq [m : (S(V''), \psi)]$, we have $S(V) \leq [m : (S(V''), \psi)]$. If $m \in S(\mathcal{M})$, then $S(V)(m) = (\tau, \bullet)$ and $S(V')(m) = (\tau, \psi')$ for some τ and ψ'. Therefore, $S(V) \leq [m : (S(V''), \psi]$.

Rule 9 If $V \leq_{\mathcal{M}} V' \in \mathcal{C}$ and $m \in \mathcal{M} \in \mathcal{C}$, then $V' \leq [m : (V'', \circ)], V \leq [m : (V'', \bullet)] \in \mathcal{C}'$.
From $S(V) \leq_{S(\mathcal{M})} S(V')$ and $m \in S(\mathcal{M})$, by the definition of $\leq_M$, $S(V)(m) = (\tau, \bullet)$ and $S(V')(m) = (\tau, \psi)$ for some τ and ψ. Since V'' is fresh, we can let $S(V'') = \tau$. Therefore, $S(V) \leq [m : (S(V''), \bullet)]$ and $S(V) \leq [m : (S(V''), \circ]$.

Rule 11 If $V \leq_{\mathcal{M}} V', V \leq [m : (V'', \bullet)] \in \mathcal{C}, \mathcal{C} \longrightarrow_A \mathcal{C}$, and $(m \in \mathcal{M}) \notin \mathcal{C}$, then $V' \leq [m : (V'', \bullet)] \in \mathcal{C}'$. Since $\mathcal{C}$ is $\mathsf{AClosed}$ and $(m \in \mathcal{M}) \notin \mathcal{C}$, $m \notin S(\mathcal{M})$. From $S(V) \leq_{S(\mathcal{M})} S(V')$, by the definition of $\leq_M$, $S(V)(m) = S(V')(m)$. Also from $S(V) \leq [m : S(V'', \bullet)]$, we have $S(V') \leq [m : (S(V''), \bullet)]$.

By the induction, we know that if $\mathcal{C}$ is satisfiable, then so is $\mathsf{Closure}(\mathcal{C})$. It is clear from the definitions of constraint satisfiability and consistency that $\mathsf{Closure}(\mathcal{C})$ is consistent as well. $\square$

Theorem 4.3 *Given a program P where $E \vdash_{\mathsf{inf}} P \mid \mathcal{C}$, P is typable iff $\mathsf{Closure}(\mathcal{C})$ is consistent.*

Proof. By Lemma 4.2 and 4.1, we know that $\mathcal{C}$ is satisfiable iff $\mathsf{Closure}(\mathcal{C})$ is consistent. Thus, we only need to show that P is typable iff $\mathcal{C}$ is satisfiable.

We first show that if $\mathcal{C}$ is satisfiable, then $\exists \Gamma$ such that $\Gamma \vdash P$.

Let S be a solution to $\mathcal{C}$ and $S(E) = \{z \mapsto \tau \mid \forall z \in \mathsf{dom}(E),\ \tau = S(E(z))\}$, where $S(V \to V') = S(V) \to S(V')$. Since the inference rules have the same structure as the typing rules, it is clear that $S(E) \vdash P$. Specifically, if $E \vdash_{\mathsf{inf}} s \parallel E' \mid \mathcal{C}'$, then $S(E) \vdash s : S(V) \parallel S(E')$.

Next we show that if $\Gamma \vdash P$, then $\mathcal{C}$ is satisfiable. If $E \vdash_{\mathsf{inf}} P \mid \mathcal{C}$, we can construct a solution to each variable in $\mathcal{C}$. For each $z \in \mathsf{dom}(E)$, let $S(E(z)) = \Gamma(z)$ and for each $F \in \mathsf{dom}(E)$, if $E(F) = V \to V'$ and $\Gamma(F) = \tau \to t$, then $S(V) = \tau$ and $S(V') = t$. If $E \vdash_{\mathsf{inf}} s \parallel E' \mid \mathcal{C}'$ and $\Gamma \vdash s \parallel \Gamma'$, for each $z \in \mathsf{dom}(E)$ and if $E(z) \in \mathcal{C}'$, then let $S(E(z)) = \Gamma(z)$. For each $z \in \mathsf{dom}(E')$ and if $E'(z) \in \mathcal{C}'$, then let $S(E'(z)) = \Gamma'(z)$. Also, $S(E(\mathsf{l})) = \Gamma(\mathsf{l})$ and $S(E'(\mathsf{l})) = \Gamma'(\mathsf{l})$. $\square$

E. Proof for the extension to allow strong update

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.

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

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

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

The judgment $\Sigma, \Gamma \vdash o : \varsigma$ says that each member 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.

$$\frac{\Sigma(\iota) \leq \varsigma}{\Sigma, \Gamma \vdash \iota : \varsigma}$$

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'.

$$\frac{\varsigma(m) = (\tau, \psi) \Leftrightarrow (\varsigma'(m) = (\tau, \psi') \ \wedge\ (\psi = \psi' \ \vee\ \bullet \leq \psi \leq \psi')}{\varsigma \leq \varsigma'}$$

Lemma E.1. *If $\Sigma, \Gamma \vdash H, \chi$ and $\Gamma \vdash z : \tau \parallel \Gamma'$, then $\exists \Sigma'$ such that*

1. $\Sigma', \Gamma' \vdash \chi(z) : \tau$
2. $\Sigma', \Gamma' \vdash H, \chi$, *and*
3. $\forall \iota \in \mathsf{dom}(\Sigma)$, $\Sigma'(\iota) \leq \Sigma(\iota)$.

Proof. By the definition of $\Gamma \vdash z : \tau \parallel \Gamma'$, either $\Gamma(z) \leq \tau$ and $\Gamma' = \Gamma$, or $\Gamma(z) = \varsigma \downarrow \varsigma', \varsigma' \leq \tau$, and $\Gamma' = [\varsigma'/\varsigma]\Gamma$.

The former case is trivial because $\Gamma(z) \leq \tau$ and $\Sigma' = \Sigma$.

For the latter case, let $\chi(z) = \iota$. From $\Sigma, \Gamma \vdash H, \chi$, we have $\Sigma(\iota) \leq \varsigma$, which means that the only difference between them is that some definite members of $\Sigma(\iota)$ are potential in ς. Therefore, we can find a ς'' such that $\Sigma' = \Sigma[\iota \mapsto \varsigma'']$ where $\Sigma(\iota) \downarrow \varsigma''$ and $\varsigma'' \leq \varsigma' \leq \tau$. By the definition of $\downarrow$, if $\varsigma \leq t$ for some t, then $\varsigma'' \leq t$. Also from $\Sigma, \Gamma \vdash H, \chi$, if $\Gamma(y) = \iota$, then either $\Gamma(y) = t$ for some t or $\Gamma(y) = \varsigma$. Thus, $\Sigma', \Gamma' \vdash \chi(y) : \Gamma(y)$ if $\chi(y) = \iota$.

For each ι' where $H(\iota')(m) = \iota$, if $\Sigma(\iota')(m) = (t, \_)$, then $\Sigma(\iota) \leq t$. From $\varsigma \downarrow \varsigma'$ and $\Sigma(\iota) \downarrow \varsigma''$, we have $\varsigma'' \leq t$. Thus, $\Sigma', \Gamma' \vdash H(\iota') : \Sigma'(\iota')$.

Finally, from $\varsigma \downarrow \varsigma, \varsigma'$ and ς have the same set of members that are definite or labeled with $*$. Thus, $\Sigma', \Gamma' \vdash H(\iota) : \varsigma'$. $\square$

We now show that a well-typed program does not access undefined object members.

Lemma E.2. *If $\Sigma, \Gamma \vdash H, \chi$ and $\Gamma \vdash s \;\|\; \Gamma'$, then $H, \chi, s \not\rightsquigarrow$ error, and if $H, \chi, s \rightsquigarrow H', \chi'$, then $\exists\, \Sigma'$ such that $\Sigma', \Gamma' \vdash H', \chi'$.*

Proof. We prove by induction and only show the three cases where the type rules are changed. For this proof, we need an additional invariant that for each $\iota \in \mathsf{dom}(\Sigma)$, if $\nexists y \in \mathsf{dom}(\chi)$ that $\chi(y) = \iota$ and $\Gamma(y) = \varsigma$, then $\Sigma'(\iota) \leq \Sigma(\iota)$.

T-New Let s be $x = \mathsf{new}\ F(z)$ and $\chi(F) = \mathsf{function}\ F(x')\{s'\}$. From Rule (T-Ctr), $\exists\ \varsigma_{\mathtt{this}}, \tau_{arg}$ such that $\Gamma_1 = \Gamma_{\mathtt{init}}[\mathtt{this} \mapsto \varsigma_{\mathtt{this}}, x' \mapsto \tau_{arg}]$, $\Gamma_1 \vdash s \;\|\; \Gamma_2$, $\Gamma_2(\mathtt{this}) \downarrow \varsigma_{res}$, and $\Gamma(F) = \tau_{arg} \to \varsigma_{res}$. By Rule (T-New), $\Gamma \vdash z : \tau_{arg} \;\|\; \Gamma''$, $\Gamma' = \Gamma''[x \mapsto \varsigma]$, and $\varsigma_{res} \downarrow \varsigma$.
Let ι be a new object label, $\chi_1 = \{\mathtt{this} \mapsto \iota, x' \mapsto \chi(z), \mathsf{ft} \mapsto \chi(\mathsf{ft})\}$, and $H_1 = H[\iota \mapsto []]$.
From Lemma E.1, $\exists \Sigma''$ such that $\Sigma'', \Gamma'' \vdash H, \chi$ and $\Sigma'', \Gamma'' \vdash \chi(z) : \tau_{arg}$. From $\mathsf{def}(\varsigma_{\mathtt{this}}) = \emptyset$, it is clear that $\Sigma''[\iota \mapsto \varsigma_{\mathtt{this}}], \Gamma_1 \vdash H_1, \chi_1$. By the induction hypothesis, there exists Σ_2, Γ_2 such that if $H_1, \chi_1, s \rightsquigarrow H_2, \chi_2$, then $\Sigma_2, \Gamma_2 \vdash H_2, \chi_2$. By the induction hypothesis and Lemma E.1, for each $\iota \in \mathsf{dom}(\Sigma)$, since there does not exists y of singleton type with $\chi_1(y) = \iota$, we have $\Sigma_2(\iota) \leq \Sigma(\iota)$. Therefore, $\Sigma_2, \Gamma'' \vdash H_2, \chi$. Since $\Sigma'(\iota) \leq \Gamma_2(\mathtt{this}) \downarrow \varsigma_{res} \downarrow \varsigma = \Gamma'(x)$ and x is the only variable of singleton type that points to ι, we have $\Sigma', \Gamma' \vdash H_2, \chi'$. where $\Sigma' = \Sigma_2[\iota \mapsto \varsigma]$ and $\chi' = \chi[x \mapsto \iota]$.

T-Upd Let s be $y.m = z$. By Rule T-Upd, we have $\Gamma \vdash z : \tau \;\|\; \Gamma'$. We only consider the case that $\Gamma'(y) = \varsigma_y$, $\Gamma'' = [\varsigma'_y/\varsigma_y]\Gamma'$, and $\varsigma'_y \leq_{(m,\tau)} \varsigma_y$. By Rule (R-Upd), if $H(\chi(y)) = o$, then $H' = H[\chi(y) \mapsto o[m \mapsto \chi(z)]]$. From $\Gamma \vdash z : \tau \;\|\; \Gamma'$ and by Lemma E.1, $\exists \Sigma'$ such that $\Sigma', \Gamma' \vdash H, \chi$.
Let $\Sigma'(\chi(y)) = \varsigma$, $\varsigma' \leq_{(m,\tau)} \varsigma$, and $\Sigma'' = [\varsigma'/\varsigma]\Sigma'$. From $\Sigma', \Gamma' \vdash H, \chi$, we have $\varsigma \leq \varsigma_y$. It is clear that $\varsigma' \leq \varsigma'_y$. Therefore, $\Sigma'', \Gamma'' \vdash H', \chi$.

T-Invk Let s be $x = y.m(z)$. By Rule T-Invk, $\Gamma(y) \leq [m : (t_{y.m}, \bullet)]$, $t_{y.m} \leq (t_{\mathtt{this}}, M) \times \tau_{arg} \to \tau_{res}$, and $\Gamma \vdash z : \tau_{arg} \;\|\; \Gamma'$. We only consider the case that $\Gamma'(y) = \varsigma_y$, $\Gamma' \vdash y : \mathtt{this} \;\|\; \Gamma_1, \varsigma'_y \leq_M \Gamma_1(y)$, and $\Gamma'' = [\varsigma'_y/\Gamma_1(y)]\Gamma_1$. By Lemma E.1, $\exists \Sigma'$ such that $\Sigma', \Gamma_1 \vdash H, \chi$.
Let $\mathsf{lookup}(f, \chi(\mathsf{ft})) = \mathsf{function}\ f(x')\{s; \mathsf{return}\ z'\}$. By Rule R-Invk, if $\chi' = \{\mathtt{this} \mapsto \chi(y), x' \mapsto \chi(z), \mathsf{ft} \mapsto \chi(\mathsf{ft})\}$ and $H, \chi', s' \rightsquigarrow H', \chi''$, then $H, \chi, x = y.m(z) \rightsquigarrow H', \chi[x \mapsto \chi''(z')]$. Similar to the proof for Lemma 3.1, we can show that $\exists \Sigma'', \Gamma_2$ such that $\Sigma'', \Gamma_2 \vdash H', \chi''$. By the induction hypothesis, $\forall \iota \in \mathsf{dom}(\Sigma')$, $\Sigma''(\iota) \leq \Sigma'(\iota)$. Together with $\varsigma'_y \leq_M \Gamma_1(y)$ and $\Gamma'' = [\varsigma'_y/\Gamma_1(y)]\Gamma_1$, we have $\Sigma'', \Gamma'' \vdash H', \chi[x \mapsto \chi''(z')]$.

□

We now show that the constraint set generated from a program is satisfiable iff its closure is consistent.

Lemma E.3. *If $E \vdash_{\mathsf{inf}} P : \mathcal{C}'$ and $\mathcal{C} = \mathsf{Closure}(\mathcal{C}')$ is consistent, then $\mathcal{C}$ is satisfiable*

Proof. We only show that $S = \mathsf{Solution}(\mathcal{C})$ satisfies four types of constraints in $\mathcal{C}$: $\mathcal{V} \leq_{(m,V)} \mathcal{V}$, $\mathcal{V} \leq V$, $\mathcal{V} \downarrow \mathcal{V}'$, and $\mathcal{V} \leq_{\mathcal{M}} \mathcal{V}'$. The satisfiability of other types of constraints can be proved the same way as in the proof of Lemma 4.1.

Consider the constraint set $\mathcal{C}'$, we can create a graph G with each $\mathcal{V}$ as a vertex and a directed edge from $\mathcal{V}'$ to $\mathcal{V}$ if $\mathcal{V} \leq_{(m,V)} \mathcal{V}'$, $\mathcal{V} \leq_{\mathcal{M}} \mathcal{V}'$, or $\mathcal{V}' \downarrow \mathcal{V}$ is in $\mathcal{C}'$. Since $\mathcal{C}'$ is generated from a program, G is in fact a collection of simple paths. Therefore, by Rule 14, 16, and 17, if $\mathcal{V}'$ can reach $\mathcal{V}$ in G, then $\mathsf{Upper}_{\mathcal{C}}(\mathcal{V}') \subseteq \mathsf{Upper}_{\mathcal{C}}(\mathcal{V})$. Consequently, if $\mathcal{C}_1$ is the $\mathsf{AClosure}$ of $\mathcal{C}'$ and $\mathcal{C}_1 \longrightarrow_B \mathcal{C}_2$, then $\mathcal{C}_2$ is also $\mathsf{AClosed}$.

1. Suppose $\mathcal{V} \leq_{(m,V)} \mathcal{V}' \in \mathcal{C}$. We need to show $S(\mathcal{V}) \leq_{(m,S(V))} S(\mathcal{V}')$. As explained above, $\mathcal{V}$ is a vertex on a simple path in G. Also, by the type inference rules, $\mathcal{V}$ may not be in another constraint of the form $\mathcal{V} \leq V$. Thus, $\mathcal{V}' \leq [m' : \_] \in \mathcal{C}$ iff $\mathcal{V} \leq [m' : \_] \in \mathcal{C}$ for any $m' \neq m$.
Now we consider three cases. The first case is when m is not a member of $S(\mathcal{V}')$. By Rule 13, $\mathcal{V} \leq [m : (V, *)] \in \mathcal{C}$. Then, $S(\mathcal{V}) = (S(V), *)$. The second case is when $S(\mathcal{V}')(m) = (S(V'), *)$. In this case, $\mathcal{C}$ does not have constraint of the form $\mathcal{V}' \leq [m : (V', \circ)]$ and therefore, Rule 14 does not add constraint of the form $\mathcal{V} \leq [m : (V', \circ)]$ to $\mathcal{C}$ and $S(\mathcal{V})(m) = (S(V), *)$. The last case is when $S(\mathcal{V}')(m) = (S(V), \psi)$ where $\bullet \leq \psi$. By the definition of $\mathsf{Solution}$, $\mathcal{C}$ contains constraint of the form $\mathcal{V}' \leq [m : (V', \circ)]$ and Rule 14 adds $\mathcal{V} \leq [m : (V', \circ)]$ to $\mathcal{C}$. In this case, $S(\mathcal{V})(m) = (S(V), \bullet)$. Therefore, $S(\mathcal{V}) \leq_{(m,S(V)} S(\mathcal{V}')$.
2. Suppose $\mathcal{V} \leq V \in \mathcal{C}$. If $S(V)(m) = (S(V'), \psi)$, there exists a constraint of the form $V \leq [m : (V', \psi)]$ in $\mathcal{C}$. By Rule 15, $\mathcal{V} \leq [m : (V', \psi)]$ and $\mathcal{V} \leq [m : (V', \circ)]$ are in $\mathcal{C}$. Thus, $S(\mathcal{V})(m) = (S(V'), \psi')$ and $\bullet \leq \psi' \leq \psi$.
3. Suppose $\mathcal{V} \downarrow \mathcal{V}'$ be in $\mathcal{C}$. By Rule 16, if $\mathcal{V} \leq [m : (V, \psi)] \in \mathcal{C}$, then $\mathcal{V}' \leq [m : (V, \psi)] \in \mathcal{C}$. As explained above, $\mathcal{V}'$ is a vertex on a simple path in G. By the inference rules, $\mathcal{V}'$ may also appear in a constraint of the form $\mathcal{V}' \leq V'$. But by the consistency rules, $V' \leq [m : (V, *)] \notin \mathcal{C}$. Thus, $\mathcal{V} \leq [m : (V, *)] \in \mathcal{C}$ iff $\mathcal{V}' \leq [m : (V, *)] \in \mathcal{C}$. By Rule 21, if $\mathcal{V}' \leq [m : (V, \bullet)] \in \mathcal{C}$, then $\mathcal{V} \leq [m : (V, \bullet)] \in \mathcal{C}$.
Therefore, $\forall m \in \mathsf{dom}(S(\mathcal{V}'))$, either $S(\mathcal{V})(m) = (\tau, \psi)$, $S(\mathcal{V}')(m) = (\tau, \psi')$, $\psi \leq \psi' \leq \bullet$ or $\psi = \psi' = \circ$, or $S(\mathcal{V})(m) = \mathsf{undef}$, $S(\mathcal{V}')(m) = (\tau, \circ)$. Thus, $S(\mathcal{V}) \downarrow S(\mathcal{V}')$.
4. Suppose $\mathcal{V} \leq_{\mathcal{M}} \mathcal{V}' \in \mathcal{C}$. As explained above, $\mathcal{V}$ is on a simple path in G. Also, by the inference rules, $\mathcal{V}$ does not appear in a constraint of the form $\mathcal{V} \leq V$. Moreover, the constraints of the form $\mathcal{V} \leq [m : (V, *)]$ are added only by Rule 13, while the constraints of the form $\mathcal{V} \leq [m : (V, \circ)]$ are added only by Rule 15. Thus, by Rule 17, $\mathcal{V} \leq [m : (V, \psi)]$ iff $\mathcal{V}' \leq [m : (V, \psi)]$ where $\psi = *$ or $\psi = \circ$. Similar to Lemma D.1, we can show that if $m \in \mathcal{M}$ and $\mathcal{V}_{\mathcal{M}}\mathcal{V}'$ are in $\mathcal{C}$, then $\mathcal{V}' \leq [m : (V, \circ)]$. for some V. Thus, by Rule 18 and 19, for $m \notin S(\mathcal{M})$, $\mathcal{V} \leq [m : (V, \bullet)] \in \mathcal{C}$ iff $\mathcal{V} \leq [m : (V, \bullet)] \in \mathcal{C}$, and for $m \in S(\mathcal{M})$, $\mathcal{V} \leq [m : (V, \bullet)] \in \mathcal{C}$ iff $\mathcal{V}' \leq [m : (V, \circ)] \in \mathcal{C}$. Thus, $S(\mathcal{V}) \leq_{S(\mathcal{M})} S(\mathcal{V}')$.

□

Lemma E.4. *If $\mathcal{C}$ is satisfiable, then $\mathsf{Closure}(\mathcal{C})$ is consistent.*

Proof. We show that if $\mathcal{C}$ is satisfiable, then its closure is also satisfiable, which implies consistency. For this, we need to prove that if S is a satisfiable solution to $\mathcal{C}$, and $\mathcal{C} \longrightarrow_A \mathcal{C}'$ or $\mathcal{C} \longrightarrow_B \mathcal{C}'$, then S is also a solution to $\mathcal{C}'$. We will only consider the additional closure rules for the extended type system.

Rule 12 If $\{\mathcal{V} \leq [m : (V, \_)], \mathcal{V} \leq [m : (V', \_)]\} \subseteq \mathcal{C}$, then $\{V \leq V', V' \leq V\} \subseteq \mathcal{C}'$. Since S solves $\mathcal{C}$, $S(V) = S(V')$, which implies $S(V) \leq S(V')$ and $S(V') \leq S(V)$.

Rule 13 If $\mathcal{V} \leq_{(m,V)} \mathcal{V}' \in \mathcal{C}$, then $\mathcal{V} \leq [m : (V, *)] \in \mathcal{C}'$. By definition, $S(\mathcal{V}) \leq_{(m,S(V))} \mathcal{V}'$ implies $S(\mathcal{V})(m) = (S(V), \psi)$ where $\psi \leq \bullet$. Thus, $S(\mathcal{V}) \leq [m : (S(V), *)]$.

Rule 14 If $\mathcal{V} \leq_{(m,V)} \mathcal{V}'$ and $\mathcal{V}' \leq [m' : (V', \psi)]$ are in $\mathcal{C}$, then $\mathcal{V} \leq [m' : (V', \psi)] \in \mathcal{C}'$ where $m' \neq m$ or $\psi = \circ$. Suppose $m' \neq m$. Since $S(\mathcal{V}) \leq_{(m,S(V))} S(\mathcal{V}')$ and $S(\mathcal{V}') \leq [m' : (S(V'), \psi)]$, $S(\mathcal{V})(m') = S(\mathcal{V}')(m') = (S(V'), \psi)$.

Suppose $\psi = \circ$. Then, $S(\mathcal{V})(m') = (S(V'), \psi')$ and $\bullet \le \psi'$. Therefore, $S(\mathcal{V}) \le [m' : (S(V'), \psi)]$.

Rule 15 If $\mathcal{V} \le V$ and $V \le [m : (V', \psi)]$ are in $\mathcal{C}$, then $\mathcal{V} \le [m : (V', \psi)]$ and $\mathcal{V} \le [m : (V', \circ)]$ are in $\mathcal{C}'$. Since $\S(\mathcal{V}) \le S(V) \le [m : (S(V'), \psi)]$, we have $\psi \ne *$ and $S(\mathcal{V}) \le [m : (S(V'), \psi)]$ and $S(\mathcal{V}) \le [m : (S(V'), \circ]$.

Rule 16 If $\mathcal{V} \downarrow \mathcal{V}'$ and $\mathcal{V} \le [m : (V, \psi)]$ are in $\mathcal{C}$, then $\mathcal{V}' \le [m : (V, \psi)] \in \mathcal{C}'$. Since $S(\mathcal{V}) \downarrow S(\mathcal{V}')$, if $S(\mathcal{V})(m) = (S(V), \psi_1)$, then $S(\mathcal{V}')(m) = (S(V), \psi_2)$ and $\psi_1 \le \psi_2 \le \bullet$ or $\psi_1 = \psi_2 = \circ$. Thus, $S(\mathcal{V}') \le [m : (S(V), \psi)]$.

Rule 17 If $\mathcal{V} \le_{\mathcal{M}} \mathcal{V}'$ and $\mathcal{V}' \le [m : (V, \psi)]$ are in $\mathcal{C}$, then $\mathcal{V} \le [m : (V, \psi)] \in \mathcal{C}'$. Since $S(\mathcal{V}) \le_{S(\mathcal{M})} S(\mathcal{V}')$, for $m \notin S(\mathcal{M})$, $S(\mathcal{V})(m) = S(\mathcal{V}')(m)$, and for $m \in S(\mathcal{M})$, $S(\mathcal{V})(m) = (\tau, \bullet)$, $S(\mathcal{V}')(m) = (\tau, \psi')$, and $\bullet \le \psi'$. Thus, $S(\mathcal{V}) \le [m : (S(V), \psi)]$.

Rule 18 If $\mathcal{V} \le_{\mathcal{M}} \mathcal{V}'$, $m \in \mathcal{M}$, and $\mathcal{V}' \le [m : (V, \circ)]$ are in $\mathcal{C}$, then $\mathcal{V} \le [m : (V, \bullet)] \in \mathcal{C}'$. Since $S(\mathcal{V}) \le_{S(\mathcal{M})} S(\mathcal{V}')$ and $m \in S(\mathcal{M})$, $S(\mathcal{V})(m) = (\tau, \bullet)$. Therefore, $S(\mathcal{V}) \le [m : S(V), \bullet)]$.

Rule 19 If $\mathcal{C} \longrightarrow_A \mathcal{C}$, $\{\mathcal{V} \le_{\mathcal{M}} \mathcal{V}', \mathcal{V} \le [m : (V, \bullet)]\} \in \mathcal{C}$, and $\{m \in \mathcal{M}\} \not\subseteq \mathcal{C}$, then $\mathcal{V}' \le [m : (V, \bullet)] \in \mathcal{C}'$. Since $S(\mathcal{V}) \le_{S(\mathcal{M})} S(\mathcal{V}')$ and $m \notin S(\mathcal{M})$, $S(\mathcal{V})(m) = S(\mathcal{V}')(m)$. Therefore, $S(\mathcal{V}) \le [m : (S(V), \bullet)]$ implies $S(\mathcal{V}') \le [m : (S(V), \bullet)]$. Also, as shown earlier, $\mathcal{C}'$ remains $\mathsf{AClosed}$ so that the closure of $\mathcal{C}'$ will not add $m \in \mathcal{M}$, which could invalidate S as a satisfiable solution.

Rule 20 If $\mathcal{C} \longrightarrow_A \mathcal{C}$, $\{\mathcal{V} \le_{(m,V)} \mathcal{V}', \mathcal{V} \le [m' : (V', \bullet)]\} \in \mathcal{C}$, then $\mathcal{V}' \le [m' : (V', \bullet)] \in \mathcal{C}'$ where $m' \ne m$. Since $S(\mathcal{V}) \le_{(m,S(V)} S(\mathcal{V}')$ and $m' \ne m$, $S(\mathcal{V})(m') = S(\mathcal{V}')(m')$. Thus, $S(\mathcal{V}') \le [m' : (S(V'), \bullet)]$.

Rule 21 If $\mathcal{C} \longrightarrow_A \mathcal{C}$, $\{\mathcal{V} \downarrow \mathcal{V}', \mathcal{V}' \le [m : (V, \bullet)]\} \in \mathcal{C}$, then $\mathcal{V} \le [m : (V, \bullet)] \in \mathcal{C}'$. Since $S(\mathcal{V}) \downarrow S(\mathcal{V}')$ and $S(\mathcal{V}') \le [m : (S(V), \bullet)]$, $S(\mathcal{V})(m) = (S(V), \psi)$ and $\psi \le \bullet$. Thus, $S(\mathcal{V}) \le [m : (S(V), \bullet)]$.

$\square$