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 addition 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-inflicted 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 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.

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

**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 the 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. Whether the member is possibly defined or definitely exists respectively. The labels are inferred along with the types of a program.

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

\[ t_x = [\text{id} : (\text{int}, \bullet), \text{set} : (t_{\text{setter}}, \bullet), \text{handle} : (t_{\text{handler}}, o)] \]

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

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

Notice that the handle method of \( t_s \) 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

\[ t'_x = [\text{id} : (\text{int}, \bullet), \text{set} : (t_{\text{setter}}, \bullet), \text{handle} : (t_{\text{handler}}, \bullet)] \]

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

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

\[ (t_0, M) \times \tau_1 \rightarrow \tau_2 \]

where \( t_0 \) is the type of this pointer, \( \tau_1 \) and \( \tau_2 \) are parameter and return type respectively. The meta variable \( \tau \) 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

\[ t_{\text{setter}} = (t_0, \{\text{handle}\}) \times t_{\text{handler}} \rightarrow \text{int} \]

where \( t_0 \) and \( t_{\text{handler}} \) are defined as

\[ t_0 = [\text{handle} : (t_{\text{handler}}, o)] \]

\[ t_{\text{handler}} = (t'_0, \emptyset) \times \text{int} \rightarrow \text{int} \]

\[ t'_0 = [\text{id} : (\text{int}, \bullet), \text{set} : (t_{\text{setter}}, \bullet), \text{handle} : (t_{\text{handler}}, \bullet)] \]

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 \( (t_0, M) \times \tau_1 \rightarrow \tau_2 \) calls function \( g \) of the type \( (t'_0, M') \times \tau'_1 \rightarrow \tau'_2 \) 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 \( \xi \), where we label object members that can receive strong update with \( + \). As shown below, \( \xi \) is the return type of the constructor `F` and \( \xi \otimes 1 \) and \( \xi \otimes 2 \) are the types of `x1` and `x2` after the last assignment.

\[ \xi = @[a : (\text{int}, +), b : (\text{string}, +)] \]

\[ \xi \otimes 1 = @[a : (\text{int}, +), b : (\text{bool}, +), c : (\text{int}, +)] \]

\[ \xi \otimes 2 = @[a : (\text{int}, +), b : (\text{string}, +), c : (\text{bool}, +)] \]

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`.

```javascript
1 function f(y) {
  2   y.a = 1;
  3   return y;
  4 }
  5 x = new F(0);
```
\[ y = x.m \ (1); \]

If the type of \( x \) starts with \( \varsigma \), then it becomes \( \varsigma' \) after the call.

\[ \varsigma = @\{ a : (int, \ast), b : (\text{string, \ast}) \}; \]
\[ \varsigma' = @\{ a : (int, \bullet), b : (\text{string, \ast}) \}; \]

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

\[ \varsigma'' = @\{ a : (\text{int, \ast}), b : (\text{bool, \ast}), c : (\text{int, \ast}) \}. \]

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.

```javascript
1 function f(y) {
2 \hspace{1em} y.c = 1;
3 return y;
4 }
5 x = new F(0);
6 z = f(x);
7 x.c = 2;
}
```

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

\[ \varsigma = @\{ a : (\text{int, \ast}), b : (\text{string, \ast}) \}; \]
\[ \varsigma' = @\{ a : (\text{int, \bullet}), b : (\text{string, \ast}), c : (\text{int, \ast}) \}. \]

Notice that \( \varsigma' \) 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.

```javascript
1 function G(i) {
2 \hspace{1em} this.a = i;
3 \hspace{1em} this.m = g;
4 }
5 function g(j) {
6 \hspace{1em} 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

\[ \varsigma = @\{ a : (\text{int, \ast}), m : (t_\ast, \ast) \}. \]

where \( t_\ast = (t, \{ b \} \times \text{int} \to t', t = [b : (\text{int, o})], \) and \( t' = [b : (\text{int, \bullet})]. \) After line 10, the type of \( x \) becomes

\[ \varsigma' = @\{ a : (\text{int, \ast}), m : (t_\ast, \ast), b : (\text{int, \bullet}) \}. \]

The variable \( y \) 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 \( s \).

\[
\begin{align*}
P &::= F_1^<\{ \ast \} \ s \\
F_1 &::= \text{function } f(x)\{s; \text{return } z\} \quad \text{function} \\
\ s &::= x \quad \text{variable declaration} \\
&\ | \quad x = z \quad \text{assignment} \\
&\ | \quad x = \text{new } F(z) \quad \text{new object} \\
&\ | \quad x = y.m \quad \text{member select} \\
&\ | \quad y.m(z) \quad \text{member call} \\
&\ | \quad s.m = z \quad \text{member update/add} \\
&\ | \quad s; s' \quad \text{sequence} \\
y &::= x \quad \text{variables} \\
&\ | \quad this \quad \text{self reference} \\
z &::= y \quad \text{function identifier} \\
&\ | \quad f \quad \text{integer} \\
&\ | \quad n \quad \text{integer}
\end{align*}
\]

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 \( r \) ranges over them.

\[
\begin{align*}
r &::= t \quad \text{name of function and object types} \\
&\ | \quad \text{top super type of function and object types} \\
&\ | \quad \text{int} \quad \text{integer} \\
\end{align*}
\]

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

\[
\begin{align*}
t &::= [m_1 : (r_1, \psi_1)]^{\{ \ast \} \ast} \quad \text{object type} \\
t &::= (t_0, M) \times t_1 \to t_2 \quad \text{function type} \\
\psi &::= \circ \quad \text{potential} \\
&\ | \quad \bullet \quad \text{definite}
\end{align*}
\]

The meta variable \( \psi \) ranges over the label \( \circ \) and \( \bullet \), which indicates whether a member is potentially or definitely present. In a function type \((t_0, M) \times t_1 \to t_2, t_0 \) is the type of this, which is always an object type while the parameter and return type \( t_1 \) and \( t_2 \) 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.
  \[ \forall t \leq \top \quad \forall \tau \leq \sigma \quad \tau \leq \sigma' \leq \sigma'' \quad \frac{\tau \leq \sigma'}{\tau \leq \sigma''} \]

- 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.
  \[ t = (t_0, M) \times \tau_1 \rightarrow \tau_2 \quad t' = (t_0, M) \times \tau_1 \rightarrow \tau_2 \quad \frac{\tau \leq \tau'}{t \leq t'} \]

- 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.
  \[ t = (t_0, M) \times \tau_1 \rightarrow \tau_2 \quad t = (t_0, M) \times \tau_1 \rightarrow \tau_2 \]

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.

\[ t(m) = \{ \tau, \psi \} \quad \frac{\forall m, t'(m) = (\tau, \psi') \Rightarrow (t(m) = (\tau, \psi) \land \psi \leq \psi')}{t \leq t'} \]

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

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

- 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 \( t \).

\[ \forall m \notin M. t(m) = t'(m) \quad \forall m \in M. t(m) = (\tau, \bullet) \land t'(m) = (\tau, \psi) \quad \frac{t \leq_M t'}{t \leq_M t'} \]

### 3.2.1 Type rules for functions and constructors

We use the symbol \( \Gamma \) to represent type environment that maps variables, function/constructor names, and constants to their types. Assume \( \Gamma(n) = \text{int} \) for any integer constant \( n \). For any variable or name of function/constructor in the domain of \( \Gamma \), we define

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

In Rule (T-Fn), we also use \( \Gamma \) 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 \( \Gamma \vdash s \quad \Gamma' \) asserts that the statement \( s \) is well-typed with the environment \( \Gamma \) and the execution of \( s \) will result in a (possibly new) environment \( \Gamma' \).

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

A function \( f \) is well-typed given an environment \( \Gamma \) 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.

\[ \Gamma \vdash P, \forall i \in 1..n \quad \Gamma \vdash s \quad \Gamma' \]

\[ \Gamma \vdash F, n \vdash l \vdash s \quad \Gamma' \]

\[ \Gamma(f) \leq (l, M) \times \tau \rightarrow \tau' \]

\[ \Gamma' = \Gamma[\text{this} \mapsto t, x \mapsto \tau, l \mapsto \emptyset] \quad \Gamma' \vdash s \quad \Gamma'' \]

\[ \Gamma'(\text{this}(z)) \leq \tau' \quad M = \Gamma''(l) \]

\[ \Gamma \vdash \text{function } f(x){s; \text{return } z} \]

\[ \Gamma(F) = \tau \rightarrow t \quad \Gamma' = \Gamma[\text{this} \mapsto l_0, x \mapsto \tau, l \mapsto \emptyset] \quad \Gamma' \vdash s \quad \Gamma'' \]

\[ \Gamma''(\text{this}) \leq t \quad \text{def}(l_0) = \emptyset \]

\[ \Gamma \vdash \text{function } F(x){s} \]

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

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

\[ \text{def}(t) = \{ m \mid t(m) = (\tau, \bullet) \} \]

#### 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 \( \text{dom} \) 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 \( y.m = z \). 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-Setl) requires the selected member to be definite, i.e. with the label \( \bullet \).

Rule (T-Inv) 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 \( \text{this} = m_j = z \) and \( x = \text{this} = m_j(z) \). Rule (T-Upd) and (T-Inv) also update the special variable \( l \) in the type environment \( \Gamma \).

For example, consider the following program fragment:

```plaintext
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, \( \Gamma(l) = \emptyset \), then after they are executed, \( \Gamma(l) = \{ m1, m2, m3 \} \).

#### 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.
Similarly, we can select a member from an object value through an object reference or a property name. A value which maps member names to values. A value is either an object or a primitive value. Moreover, lookup(F, \text{fn}) = \text{fn}_{j} if \text{fn}_{j} is the declaration of the function F and lookup(F, F\text{fn}_{i}) = \text{fn}_{j} if \text{fn}_{j} is the declaration of the constructor F, where i \in [1..n]. The reduction of a statement is written in form of \text{H}, \chi, s \rightarrow \text{H'}, \chi' which means that the execution of a statement s given the configuration of a heap H and a stack \chi results in a new configuration \text{H'}, \chi'.

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 \chi and after the execution of s, x 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 any runtime error.
environment \( \Sigma \) and \( \Gamma \), where \( \Sigma \) maps object labels to their types 
\(- \Sigma = \{ \iota \mapsto t_i : \iota \in 1..n \} \).

The judgment \( \Sigma, \Gamma \vdash \tau \) asserts that the value \( v \) is well-typed with the type \( \tau \).

\[
\Sigma, \Gamma \vdash \tau \quad \text{if} \quad \Sigma(t) \leq t, \quad \Gamma(f) \leq t
\]

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 \( \Sigma, \Gamma \vdash o : \tau \) asserts that the object \( o \) is well-typed with the type \( \tau \).

\[
\forall m. f(m) = (\tau, \bullet) \Rightarrow \Sigma, \Gamma \vdash o(m) : \tau
\]

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

\[
\forall \iota, \ell \in \text{dom}(\Sigma) \Rightarrow \iota \in \text{dom}(H) \quad \forall y, y \in \text{dom}(\Gamma) \Rightarrow y \in \text{dom}(\chi) \quad \forall y \in \text{dom}(\chi), \Sigma, \Gamma \vdash H(y) : \Sigma(\ell)
\]

The judgment \( \Sigma, \Gamma \vdash H, \chi \) says that the heap \( H \) and stack \( \chi \) are well-formed with respect to the environment \( \Sigma \) and \( \Gamma \). For this invariant to hold, the domains of \( H \) and \( \Sigma \) must be the same and the domains of \( \chi \) and \( \Gamma \) have the same set of variables; also, each object in \( H \) and each variable in \( \chi \) must be well-typed. Each function/constructor declaration in \( \chi \) is well-typed with the environment \( \Gamma_{\text{init}} \), which is defined as the environment maps function constructors/names to their types.

From the typing rules, we can show that if a well-typed function \( f \) has the type \((t, M) \times \tau_1 \rightarrow \tau_2\), then \( M \) correctly identified the added (or updated) members of the receiver-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.

Theorem 3.2 (Type Soundness). If \( \Sigma, \Gamma \vdash P \parallel \Gamma' \), then \( \emptyset, \emptyset, P \not\vdash \text{error} \) and if \( \emptyset, \emptyset, P \not\vdash H, \chi \), 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 \( E \vdash_{\text{inf}} P \parallel C \) generates a set of constraints \( C \) from a program \( P \) based on the initial environment \( E \). Likewise, \( E \) maps function and constructor names to distinct type variables. Moreover, the judgment \( E \vdash_{\text{inf}} F \) \( F \) \( C \) generates a set of constraints \( C \) from a function or constructor declaration \( F \) based on the environment \( E \).

Moreover, the variable \( M \) in Figure 7 corresponds to a set of member names and for each constructor function \( F \), we create a unique variable \( VF \) for the initial type of this pointer.

\[
E \vdash_{\text{inf}} F \parallel C, \forall i \in 1..n \quad E \vdash_{\text{inf}} s \parallel E' | C_0
\]

\[
E \vdash_{\text{inf}} F n | C, \forall i \in 1..n \quad E \vdash_{\text{inf}} s | \bigcup_{i \in 0..n} C_i
\]

\[
V_{\text{this}}, V_{\text{arg}}, V_{\text{res}}, M \text{ fresh}
\]

\[
E' = E[x \mapsto V_{\text{arg}}, \text{this} \mapsto V_{\text{this}}, l \mapsto 0] \quad E' \vdash_{\text{inf}} s \parallel E'' | C'
\]

\[
C'' = C' \cup \{ M = E'(l), E''(z) \leq V_{\text{res}}, V_{\text{this}} \leq \{ \} \}
\]

\[
C = C'' \cup \{ E(f) \leq (V_{\text{this}}, M) \times V_{\text{arg}} \rightarrow V_{\text{res}} \}
\]

\[
E \vdash_{\text{inf}} \text{function } f(x){\parallel \{ x; \text{return } z \}} \parallel C
\]

\[
E' = E[x \mapsto V_{\text{arg}}, \text{this} \mapsto V_{\text{res}}, l \mapsto 0] \quad E' \vdash_{\text{inf}} s \parallel E'' | C'
\]

\[
E(F) = V_{\text{arg}} \rightarrow V_{\text{res}}, C = C' \cup \{ V_{\text{res}} \leq \{ \}, E''(\text{this}) \leq V_{\text{res}} \}
\]

\[
E \vdash_{\text{inf}} \text{function } F(x){\parallel \{ s \}} \parallel C
\]

\[
V \text{ fresh } E' = E[x \mapsto V]
\]

\[
E \vdash_{\text{var}} \text{var } x \parallel E' \parallel \emptyset
\]

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

\[
V_{y,m}, V_y, M \text{ fresh } E' = E[y \mapsto V_y] \quad y \not\in \text{this} \Rightarrow E' = E
\]

\[
E(y) \leq m \Rightarrow V_{y,m} \leq V_{\text{res}} \leq M \quad E(y), m \in M
\]

\[
E \vdash_{\text{inf}} y, m = z \parallel E' \parallel C
\]

\[
V_{y,m} \text{ fresh } C = \{ E(y) \leq m : V_{y,m}, \bullet \} \quad E' = E[x \mapsto V_{y,m}]
\]

\[
E \vdash_{\text{inf}} x = y, m \parallel E' \parallel C
\]

\[
V_{y,m}, V_y, V_{\text{this}}, V_{\text{arg}}, V_{\text{res}}, M \text{ fresh } E' = E[y \mapsto V_y, x \mapsto V_{\text{res}}] \quad C = \{ E(y) \leq m : V_{y,m}, \bullet \}, V_{y,m} \leq (V_{\text{this}}, M) \times V_{\text{arg}} \rightarrow V_{\text{res}}
\]

\[
E(z) \leq V_{\text{arg}}, V_{\text{res}} \leq M \quad E(y), y \not\in \text{this} \Rightarrow E' = E
\]

\[
E \vdash_{\text{inf}} x = y, m(z) \parallel E' \parallel C
\]

\[
E(F) = V_{\text{arg}} \rightarrow V_{\text{res}}, C = \{ E(z) \leq V_{\text{arg}} \} \quad E' = E[x \mapsto V_{\text{res}}]
\]

\[
E \vdash_{\text{inf}} x = \text{new } F(z) \parallel E' \parallel C
\]

\[
E \vdash_{\text{inf}} s \parallel E'' \parallel C \quad E' \vdash_{\text{inf}} s \parallel E'' \parallel C
\]

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:

\[
E \vdash_{\text{inf}} s \parallel E' | C
\]

which generates a set of constraints \( C \) 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 \( V \rightarrow V' \), and it maps a special variable \( l \) to \( \{ \} \). \( M \) 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 \( \emptyset \cup M_1 \cup \ldots \cup M_n \). To simplify notation, we write \( \emptyset \cup M_1 \cup \ldots \cup M_n \) as \( M_1 \cup \ldots \cup M_n \).
After generating type constraints, we have a set of constraints of the following format:

\[\begin{align*}
\text{int} &\leq V \\
V &\leq (V_0, M) \times V_1 \rightarrow V_2 \\
V &\leq [m : (V', \psi)] \\
V' &\leq V \leq M \\
m &\in M \\
M &\in \mathcal{N}
\end{align*}\]

where \(\mathcal{N} = \emptyset \cup \bigcup_{i \in 1..n} M_i\).

### 4.1 Closure rules

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


\[
\begin{align*}
U &::= \text{int} | V \\
W &::= U | [m : (V', \psi)] | (V, M) \times V_1 \rightarrow V_2
\end{align*}
\]

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 \(M\) variables. Note that in Rule 6, the constraint \(M = \bigcup_{i \in 1..n} M_i\) can also represent constraint of the form \(M = M_1\) 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:

\[U \leq W \leq V \leq M \leq V'\]

A solution \(S\) to a constraint set \(C\) maps each \(V\) in \(C\) to int, top, or \(t\), or maps each \(M\) in \(C\) to set of member names, and

\[S(\text{int}) = \text{int} \quad S([m : (V', \psi)]) = [m : (S(V'), \psi)] \quad S((V_0, M) \times V_1 \rightarrow V_2) = (S(V_0), S(M)) \times S(V_1) \rightarrow S(V_2)\]

We say that a constraint set \(C\) is satisfiable if there exists a solution \(S\) to \(\hat{C}\) such that

\[\begin{align*}
U \leq W \in C &\Rightarrow S(U) \leq S(W) \\
V \leq M \leq V' \in C &\Rightarrow S(V) \leq S(M) \leq S(V') \\
\{m \in M\} \subseteq C &\Rightarrow \{m \in S(M)\} \\
\mathcal{M} = N \in C &\Rightarrow S(M) = S(N) \\
V_F \in \mathcal{C} \text{ appears in } C &\Rightarrow \text{def}(S(V_F)) = \emptyset
\end{align*}\]

### 4.3 Constraint closure

Before solving a constraint set \(C\), we will compute its closure.

**Definition** A constraint set \(C\) is \(\text{AClosed}\) if \(C \rightarrow_{\text{A}} C\). Let \(\text{AClosure}(C) = \hat{C}'\) if \(C \rightarrow_{\text{A}} \hat{C}'\) and \(C' \rightarrow_{\text{A}} \hat{C}\), where \(\rightarrow_{\text{A}}\) is defined by Rule 10 and \(\rightarrow_{\text{A}}\) is a transitive closure of \(\rightarrow_{\text{A}}\).

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 \(\rightarrow_{\text{A}}\) is monotone and increasing, we can always find \(\text{AClosure}\) of a constraint set.

**Definition** A constraint set \(C\) is \(\text{Closed}\) if \(C \rightarrow_{\text{B}} C\). We define \(\text{Closure}(C) = \hat{C}'\) if \(\text{AClosure}(C) \rightarrow_{\text{B}} \hat{C}'\) and \(C' \rightarrow_{\text{B}} \hat{C}\), where \(\rightarrow_{\text{B}}\) is defined by Rule 11 and \(\rightarrow_{\text{B}}\) is a transitive closure of \(\rightarrow_{\text{B}}\).

If \(C \rightarrow_{\text{B}} C'\), then \(C'\) remains \(\text{AClosed}\) 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, \(\text{Closure}(C)\) is also \(\text{AClosed}\). Since a constraint set of finite variables is bounded in size and \(\rightarrow_{\text{B}}\) is monotone and increasing, we can always find the closure of a constraint set \(C\).

### 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 \(C\) is inconsistent if one of the followings is true:

1. \([V \leq \text{int}, V \subseteq I] \subseteq C\)
2. \([V \leq \text{int}, V \subseteq [m : (V, \psi)] \subseteq C]\)
3. \([V \leq \text{int}, V \leq (V_0, M) \times V_1 \rightarrow V_2] \subseteq C\)
4. \([V \subseteq [m : (V, \psi)] \subseteq C, \forall i \in 1..n]\)
5. \([V \subseteq (V_0, M) \times V_1 \rightarrow V_2 \subseteq C]\)
6. \([V \subseteq (C \rightarrow_{\text{A}} \hat{C}) \subseteq C\]
7. \([V \subseteq (V_0, M) \times V_1 \rightarrow V_2 \subseteq C]\)

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 \(M\) variables. For example, if \(C\) is closed and \(\{m \in M' : M = M' = \bigcup_{i \in 1..n} M_i\} \subseteq C\) then by Rule 6, \(m \in M\) has to be in \(C\) and any solution \(S\) to \(\hat{C}\) will have \(m \in S(M)\) but we also want \(S(M) = \bigcup_{i \in 1..n} S(M_i)\). Thus, \(C\) is inconsistent if it does not contain \(m \in M_i\) for some \(i\).

The last rule states 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 \(V \subseteq [m : (V', \bullet)]\). If the member is not defined before the access, then the closure rules will eventually generate \(V_F \subseteq [m : (V', \bullet)]\), where \(V_F\) represents the return type of the object’s constructor, otherwise, only \(V_F \subseteq [m : (V', o)]\) will be generated.

### 4.5 Constraint solution

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

\[\text{Upper}_C(V) = \{W \mid (V \leq W) \in C\}\]

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

1. \(S(M) = \{m \mid (m \in M) \subseteq C\}\)
2. \(S(V) = \text{int} \text{ if } \int \in \text{Upper}_C(V)\)
Lemma 4.2. \( C \) is satisfiable.

To prove Lemma 4.2, we only need to show that if \( C \) is satisfiable, then its closure is also satisfiable. A satisfiable constraint set is always consistent.

\[ \text{Theorem 4.3. Given a program } P \text{ where } E \vdash_{nt} P \mid C, P \text{ is typable if } \text{Closure}(C) \text{ is consistent.} \]

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 \( \varsigma \) ranges over the singleton type names, which are distinct from the obj-type names. Each singleton type \( \varsigma \) is defined by an equation of the form

\[ \varsigma = \emptyset \{ m : (\tau, \psi) \}_{i \in 1..n}, \]

where \( \tau \) cannot be singleton types and \( \psi \) can be either *, o, or 0. 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 \( \varsigma \) defined as above, \( \varsigma(m_i) = (\tau, \psi_i) \) for all \( i \in 1..n \) and \( \varsigma(m_i) = \text{undef} \) otherwise.

5.1 Type rules

Before introducing new type rules, we first define an operator \( \downarrow \) to downgrade the singleton type \( \varsigma \) to \( \varsigma' \) so that some of the * members in \( \varsigma \) are definite in \( \varsigma' \) and \( \varsigma' \) 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.

\[ \varsigma = \emptyset \{ m : (\tau, \psi) \}_{i \in 1..n} \]

\[ \varsigma' = \emptyset \{ m : (\tau, \psi') \}_{i \in 1..n}, m : (\tau, 0) \}_{i \in n+1..m} \]

\[ \forall i \in 1..n \psi_i \leq \psi_i' \leq * \quad \forall V \psi_i' = \psi_i = o \]

where we define * \( \leq \bullet \) and * \( \leq \bullet \).
A singleton type \( \varsigma \) is a subtype of an object type \( t \) if some definite members of \( \varsigma \) appear as potential members in \( t \) while members labeled with \( \ast \) in \( \varsigma \) do not appear in \( t \).

\[
\tau (m) = (\tau, \psi) \Rightarrow (\varsigma (m) = \tau \land \ast \leq \psi \leq \psi')
\]

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 object type. This can happen when a variable of singleton type is passed to a parameter or assigned to an object.

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

```javascript
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 \( \varsigma \), and it becomes \( \varsigma' \) on line 12.

\[
\varsigma = \emptyset \{ a : (\text{int}, \ast), b : (\text{string}, \ast) \}
\]

\[
\varsigma' = \emptyset \{ a : (\text{int}, \bullet), b : (\text{string}, \ast), c : (\text{int}, o) \}
\]

\[
t_y = \emptyset \{ a : (\text{int}, o), c : (\text{int}, o) \}
\]

Before the variable \( x \) is passed to the parameter \( y \) of the function \( f \), we downgrade the type \( \varsigma \) to \( \varsigma' \) so that \( \varsigma \leq t_y \). Thus, the type of \( y \) by the update is \( \varsigma' \).

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 \( \Gamma \vdash x : \tau \parallel \Gamma' \). In particular, if the type of \( x \) is a subtype of \( \tau \), then the environment \( \Gamma \) remains the same. Otherwise, we downgrade the type of \( x \) from \( \varsigma \) to \( \varsigma' \) so that \( \varsigma' \) is a subtype of \( \tau \) and we replace all occurrences of \( \varsigma \) with \( \varsigma' \) in \( \Gamma \) with \( \varsigma' \), which is written as \( [\varsigma' / \varsigma] \Gamma \).

To see why this last step is necessary, consider the previous example where the variable \( x \) and \( z \) are aliases of each other and they both have the type \( \varsigma \). If the type of \( x \) remains \( \varsigma \) after the call \( f(x) \), then the update \( x.a = \text{true} \) 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 \( z \).

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

\[
\varsigma = \emptyset \{ m_1 : (\tau_1, \psi_1) \} \quad m_1 \neq m, \forall i \in 1..n
\]

\[
\varsigma' = \emptyset \{ m_i : (\tau_i, \psi_i) \}
\]

\[
\varsigma' \leq_{(m, \ast)} \varsigma \]

For example, the types of \( \text{this} \) at line 2 \( \varsigma_{\text{this}} \) and at line 3 \( \varsigma'_{\text{this}} \) have the relation \( \varsigma_{\text{this}} \leq_{(\text{string}, \ast)} \varsigma'_{\text{this}} \).

\[
\varsigma_{\text{this}} = \emptyset \{ a : (\text{int}, \ast) \}
\]

\[
\varsigma'_{\text{this}} = \emptyset \{ a : (\text{int}, \bullet), b : (\text{string}, \ast) \}
\]

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

\[
\varsigma(m) = (\tau, \psi) \leq \psi \leq \psi' \leq \psi(\varsigma(m) = (\tau, \psi) \leq \psi \leq \psi' \leq \psi)
\]

\[
\forall m \in M, \varsigma'(m) = (\varsigma(m), \leq \psi \leq \psi') \leq \psi \leq \varsigma \leq \varsigma M \leq \varsigma
\]

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

\[
\Gamma' = \Gamma[\text{this} \mapsto x, x : \tau_{\text{arg}}] \quad \Gamma' \vdash s \parallel \Gamma''
\]

\[
\Gamma(F) = \tau_{\text{arg}} \mapsto \varsigma_{\text{res}}, \Gamma''(\text{this}) \downarrow \varsigma_{\text{res}} \quad \text{def}(\varsigma) = \emptyset
\]

\[
\Gamma' = \text{function } F(x)[s]
\]

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 \( V \) to represent singleton types while \( V \) still represents object-types.

The inference rules generates some new types of constraints:

\[
\mathcal{V} \downarrow \mathcal{V} \quad \mathcal{V} \leq \mathcal{V} \leq_{(\mathcal{N}, \mathcal{V})} \mathcal{V} \leq_{\mathcal{M}} \mathcal{V} \leq \mathcal{M} \leq \mathcal{V} \leq \mathcal{M} : (\mathcal{V}, \psi)
\]

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
to the constraint set $C$, then

$V \leq_{(m, V)} V \in C$ $\Rightarrow$ $S(V) \leq_{(m, S(V))} S(V')$

$V \leq V'$ $\Rightarrow$ $S(V) \leq S(V')$

$V \leq_{M} V'$ $\Rightarrow$ $S(V) \leq_{S(M)} S(V')$

$V \downarrow V'$ $\Rightarrow$ $S(V) \downarrow S(V')$

$V \leq [m : (V, \psi)] \in C$ $\Rightarrow$ $S(V) \leq [m : (S(V), \psi)]$

$V_F$ appears in $C$ $\Rightarrow$ $\text{def}(S(F_V)) = 0$.

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 $V \leq_{M} V'$, which are similar to those for $V \leq_{M} V'$. 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 object types. Rule 19, 20, and 21 are applied to AClosed constraint set and the resulting constraint set is still AClosed.

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

1. $V \not\in \mathbb{C}$
2. $V \leq (V_0, V_1) \times V_2 \rightarrow V_3 \in C$
3. $V \leq [m : (\cdot, \cdot, \cdot)] \in C$
4. $V_F \leq [m : (V, \cdot)] \in C$.

where the last rule replaces $V_F \leq [m : (V, \cdot)] \in C$ in the previous consistency rules. A singleton type cannot be an integer or function type and an object type cannot have a member labeled with $\ast$ 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 $C$ for its $V$ variables so that $S(V) = [m : (\tau, \psi)]^{(1-n)}$ where $\psi_1 \in 1..n$, $\tau_i = S(V)$ for some $V$ such that $[m : (V, \cdot)] \in \text{Upper}(C)$ and

1. $\psi_i = \ast$ if $X = \{\ast\}$ or $\{\ast, \cdot\}$,
2. $\psi_i = \circ$ if $X = \{\circ\}$, and
3. $\psi_i = \bullet$ otherwise,

where $X = \{\psi : [m : (\cdot, \ast)] \in \text{Upper}(V)\}$.

The label $\psi_i$ of $S(V)$ is $\ast$ when $\ast \in X$. We also let $\psi_i$ be $\bullet$ if $X = \{\ast, \bullet\}$ since a member select statement adds a constraint of the form $V \leq [m : (V, \bullet)]$. Reading a member of a singleton type does not make the member definite. Finally, we let $\psi_i$ be $\bullet$ if $\{\ast, \circ\} \subseteq X$, 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 $V$ also exists in the type $V$ where $V \leq V$ by propagating constraints of the form $V \leq [m : (V, \circ)]$ through Rule 15. When both constraints of the form $V \leq [m : (V', \circ)]$ and $V \leq [m : (V', \circ)]$ are present, it indicates that $m_i$ has to be a definite member in $S(V)$.

### 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.
the parameter type needs to be singleton type. This may present
formalism, abstract locations are assigned to new expressions that
the types of existing members. Objects of summary types can no
receive strong updates for adding new members or even changing
promoted to the corresponding summary type when the next object
singleton type and summary type, where each singleton type is

two kinds of object types: pro-type and obj-type. A pro-type’s reservation part may be
to parameters or fields of obj-type.

extensions and encapsulation.

types without subtyping from sealed obj-types that allow width
with object extensions, which also distinguishes extensible pro-
to parameters or fields of obj-type.

loss the ability of having strong updates even after it is assigned
newly created objects to have singleton types similar to their pro-
types: pro-type and obj-type. A pro-type’s reservation part may be
in the function body. They distinguish two kinds of object
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
interface part. They define a type construct to recursively encode
extension, the extended member moves from reservation part to the
labels, they separate potential and definite members of an object
and unrestricted extensions.

This distinction allows new objects to have strong updates
after an object loses its recency.

Jensen et al. [12] have implemented a practical analyzer to de-
tect 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 pro-
gams with inheritance, assignments, and late binding. The purpose
is to guarantee all messages are understood while allowing poly-
morphic methods. The algorithm handles late binding with con-
ditional 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 pre-
cision via an incremental algorithm. Further enhancement on pre-
cision and efficiency were made by Agesen in his Cartesian Prod-
uct Algorithm [2], which was applied to type inference for Python
programs to improve compiled code [20]. Eifrig et al. [8] de-
veloped 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 Car-
tesian Product Algorithm with the ability to analyze data polymor-
phic programs.

DRuby [9] is a tool to infer types for Ruby, which is a class-
based scripting language. DRuby includes a type system with fea-

\[ V \leq [m : (V, \psi)], V' \leq [m : (V', \psi)] \rightarrow V \leq V', V' \leq V \]  
\[ V \leq_{[m, V]} V' \rightarrow V \leq [m : (V, \ast)] \]  
\[ V \leq_{[m, V]} V', V' \leq [m' : (V', \psi)] \rightarrow V \leq [m' : (V', \psi)] \text{ where } m' \neq m \text{ or } \psi = \circ \]  
\[ V \leq V, V \leq_{[m, V]} V' \rightarrow V \leq [m : (V', \psi)] \]  
\[ V \leq V', V \leq [m : (V, \psi)] \rightarrow V \leq [m : (V', \psi)], V \leq [m : (V', \psi)], V \leq [m : (V', \psi), \circ] \]  
\[ V \searrow V', V \leq [m : (V, \psi)] \rightarrow V' \leq [m : (V, \psi)] \]  
\[ V \leq_{M} V', V' \leq [m : (V, \psi)] \rightarrow [m : (V, \psi)] \]  
\[ V \leq_{M} V', V' \leq [m : (V, \circ)] \rightarrow V \leq [m : (V, \bullet)] \]  
\[ \{ V \leq_{M} V', V \leq [m : (V, \bullet)] \} \subseteq C \rightarrow \{ m \in M \} \subseteq C \]  
\[ \{ V \leq_{M} V', V \leq [m : (V, \bullet)] \} \subseteq C \rightarrow \{ m \neq m' \} \]  
\[ \{ V \downarrow V', V \leq [m : (V, \bullet)] \} \subseteq C \rightarrow \{ m : (V, \bullet) \} \]  
\[ \{ V \downarrow V', V \leq [m : (V, \bullet)] \} \subseteq C \rightarrow \{ m : (V, \bullet) \} \]  
\[ \{ V \downarrow V', V \leq [m : (V, \bullet)] \} \subseteq C \rightarrow \{ m : (V, \bullet) \} \]  
\[ \{ V \downarrow V', V \leq [m : (V, \bullet)] \} \subseteq C \rightarrow \{ m : (V, \bullet) \} \]  

Figure 11. Additional closure rules
tures 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 typeable 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: size : (int, •) \ldots, with the potential size member.

\[
\begin{align*}
    x & = \text{new Form}(); \\
    \text{addSize}(x);
\end{align*}
\]

It seems straightforward to have this type of extension so that after the call returns, the type of x becomes size : (int\cdot)\ldots. The problem is to only identify members added to the function parameter before it is overwriten 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


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 illustrates the conditions that lead to null pointer exceptions.

\[ H(\chi(y))(m_j) = \text{undef} \]
\[ H, \chi, x = y, m_j \sim \text{error} \]
\[ H, \chi, s \sim \text{error} \text{ or } (H, \chi, s \sim H', \chi' \land H', \chi', s' \sim \text{error}) \]
\[ H(\chi(y))(s); s' \sim \text{error} \]
\[ H, \chi, x = y, m_j(z) \sim \text{nullPtrEx} \]

Figure 12. Error of accessing undefined members or functions

\[ \chi(y) = \text{null} \]
\[ H, \chi, x = y, m_j \sim \text{nullPtrEx} \]
\[ \chi(y) = \text{null} \]
\[ H, \chi, y, m_j = z \sim \text{nullPtrEx} \]
\[ H, \chi, s \sim \text{nullPtrEx} \text{ or } H, \chi, s \sim H', \chi', H', \chi', s' \sim \text{nullPtrEx} \]

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.

```java
function Form(a) {
    this.set = setter;
}

function handler(c) {
    return 0;
}

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 \( V_{\text{setter}} \). The exception is \( V_{\text{Form}} \), 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 \( V_s \) are for the types of \( x \) before and after the call \( x.x.handle(1) \) while \( V_{g.x} \) is for the type of \( x \) after the call \( x.x.handle(1) \).

![Generated constraints](image)

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 \( V_{\text{Form}} \) is \( \{ |, \text{set} : (V_1, \circ) \} \) and satisfies the consistency rule that it may not contain object types with definite members.

Closure also generates some constraints for \( M \) variables with clear solutions.

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 \( \Gamma \vdash s \ | \ | \Gamma', \forall m \in \Gamma(l) \Gamma(m) \leq |m : (\_ \bullet)| \), then \( \Gamma'(\text{this}) \leq \Gamma'(\text{null}) \Gamma(\text{this}).

**Proof.** If \( \Gamma' = \Gamma \), then by definition of \( \leq_M \), \( \Gamma(\text{this}) \leq \Gamma'(\text{null}) \Gamma'(\text{this}) \) since every \( m \) in \( \Gamma(l) \) is a definite member in \( \Gamma(\text{this}) \).
\[\Gamma' \text{ may be different from } \Gamma \text{ only if } s \text{ is a member update/add or a method call on this variable, or } s \text{ is a sequence statement.}\

If \(s = \textbf{this.m}_j = z\), then \(\Gamma'(l) = \Gamma(l) \cup \{m_j\}\), \(\Gamma'(\textit{this}) \leq \{m_j \mid (\_; \_).\} \) and \(\Gamma'(\textit{this}) \leq \Gamma(l)\). If \(\Gamma'(l) = \Gamma(l) \cup M\), \(\Gamma'(\textit{this}) \leq M\). By definition of \(\leq\) and \(\Gamma'(\textit{this}) \leq \Gamma'(l)\).

If \(s = 1; s_1 \mid s_2\) and \(s_1 \neq \textit{null}\), we have \(\Gamma'(\textit{this}) \leq \Gamma'(s_1)\). By induction, \(\Gamma'(\textit{this}) \leq \Gamma'(s_1)\). By the definition of \(\leq\), we have \(\Gamma'(\textit{this}) \leq \Gamma'(l)\).

Lemma C.2. If \(\Gamma \vdash f(x) \{s; \text{return } z\}\), \(\Gamma(f) = (t, M) \times \tau_1 \rightarrow \tau_2, \Gamma' = \Gamma(l) \rightarrow t, x \rightarrow \tau_1, l \rightarrow \emptyset\), \(\Gamma' \vdash s \mid \Gamma'(t)\), and \(M = \Gamma'(l)\), then \(\Gamma'(\textit{this}) \leq \Gamma'(l)\).

**Proof.** Since \(\Gamma'(l) = \emptyset\) and \(\Gamma' \vdash s \mid \Gamma'(t)\), from Lemma C.1, \(\Gamma'(\textit{this}) \leq \Gamma'(l)\), which is \(\Gamma'(\textit{this}) \leq \Gamma(t)\). □

Lemma C.3. If \(\Sigma, \Gamma \vdash H, \chi, y(m) = \tau, \Gamma(y) = \chi\), then \(H(v)(m) = v\) where \(\Sigma, \Gamma \vdash v \rightarrow \tau\).

Lemma C.4. If \(\Sigma, \Gamma \vdash H, \chi, y \vdash \Gamma', \text{ then } H, \chi, y \vdash \tau\).

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

\(R-\text{Dec}\) Let \(s = \text{var } x\). Then \(H, \chi, x \vdash H, \chi[x \rightarrow \text{null}]\). Since null can have any type and if \(\Gamma \vdash s \mid \Gamma[l] \rightarrow \tau\), then \(\Gamma[x \rightarrow \tau] \vdash H, \chi[x \rightarrow \text{null}]\).

\(R-\text{Asn}\) Let \(s = x = y\). By Rule (T-Asn), \(\Gamma' = \Gamma[x \rightarrow \tau]\). By the induction hypothesis, we have \(\Sigma, \Gamma \vdash \chi : \tau\). Hence, \(\Sigma, \Gamma' \vdash H, \chi[x \rightarrow \tau]\).

\(R-Sel\) Let \(s = y.m_j\). By Rule (T-Sel), \(\Gamma(y) \leq \{m_j \mid (\cdot; \cdot)\}\) and \(\Gamma'(l) = \Gamma[x \rightarrow \tau]\). Then \(\exists u\) such that \(\chi(y) = u\). From \(\Gamma \vdash H, \chi \vdash \Sigma, \exists u\) such that \(H(u)(m_j) = v_j\) and \(\Sigma \vdash v_j \rightarrow \tau\). Hence, \(\Sigma, \Gamma' \vdash H, \chi[x \rightarrow v_j]\).

\(R-Upd\) Let \(s = y.m_j\). Let \(\chi(y) = u\). Then \(H, \chi, y.m_j \vdash \tau' \rightarrow \tau''\), where \(\tau'' = H(u \rightarrow H(u)(m_j) \rightarrow \chi(z))\).

By Rule (T-Upd), \(\Sigma \vdash \tau'' \rightarrow \tau''\), where \(\Sigma \vdash \tau'' \rightarrow \tau''\). Also by Rule (T-Upd), \(\Sigma \vdash H(u \rightarrow \tau)\). Hence, \(\Sigma' \vdash H(u \rightarrow \tau)\). By induction, \(\Sigma' \vdash H(u \rightarrow \tau)\). Hence, \(\Sigma' \vdash H(u \rightarrow \tau)\).

\(R-\text{Ink}\) Let \(s = y.m_j\). By Rule (T-Ink), \(\Gamma(y) \leq \{m_j \mid (t_j; \cdot)\}\) and \(t_j \leq (t_0, M) \times \tau_1 \rightarrow \tau_2\). From \(\Gamma \vdash H, \chi, \Sigma \vdash \tau_1 \rightarrow \tau_2\), \(\Sigma' \vdash \tau_1 \rightarrow \tau_2\). Thus, \(\Sigma' \vdash H, \chi, \Sigma' \vdash \tau_1 \rightarrow \tau_2\). By Rule (T-FInk), \(\Gamma_0 \vdash \tau_1 \rightarrow \tau_2\). Then by Rule (T-FInk), \(\Gamma_0 \vdash \tau_1 \rightarrow \tau_2\).

Figure 16: Constraint solution
\[ H', \chi', \gamma', \text{ then } \Sigma', \Gamma_0' \vdash H', \chi', \gamma' \text{ and } \gamma'_0'(\chi') \leq \tau_2. \text{ Let } \chi(y) = \iota, \text{ and } \Gamma_0' = [y \mapsto x, t \mapsto \tau_2], \text{ where } t \leq \tau \leq \Delta(y). \text{ By Lemma C.2, } \Gamma_0'(\text{this}) \leq \tau_0. \text{ Since } \Sigma'(y) \leq \Gamma_0'(\text{this}), \text{ each } m \in M \text{ is definite member of } \Sigma'(y). \text{ From } \Gamma_0'(\text{this}) \leq \Delta(y), \text{ the only differences between } \Gamma(y) \text{ and } \Gamma_0'(\text{this}) \text{ are the labels on members in } M. \text{ Also since } \Sigma'(y) \leq \Sigma(y) \leq \Gamma(y) \text{ it follows that } \Sigma'(y) \leq \Gamma'(y).

By the induction hypothesis, \forall \iota \in dom(\Sigma), \Sigma'(y) \leq \Sigma(y). \text{ Thus, } \Sigma', \Gamma_0' \vdash \Gamma'(y), \forall \iota \in dom(\chi). \text{ From Rule (T-Invk), we have } \Gamma'(x) = \tau_2. \text{ From } \Sigma', \Gamma_0' \vdash H', \chi, \gamma', \text{ we have } \Sigma', \Gamma_0' = \chi'(\gamma') \vdash \Gamma'(x). \text{ From } \Gamma'(x) \leq \tau_2, \text{ it follows that } \Sigma', \Gamma_0' \vdash H', \chi(x \mapsto \chi'(x')).

\textbf{R-New} \text{ Let } s \text{ be } new F(z) \text{ and function } F(x)'(z)' \text{ be the definition of } F. \text{ From Rule (T-Ctr), } \exists \tau_0 \text{ and } \tau \text{ such that } \Gamma_1 = \Gamma_1(\text{this}) \vdash \tau_0, x' \rightarrow \tau_1, \Gamma_1 \vdash s \parallel \tau_2, \Gamma_2, \Gamma_2(\text{this}) \leq \tau_0', \text{ and } \Gamma(F) = \tau \rightarrow \tau_0'. \text{ By Rule (T-New), } \Gamma'(x) = \Gamma[x \mapsto \tau_0] \text{ and } \Gamma(\chi) \rightarrow \tau. \text{ Let } \iota \text{ be a new object label, } \chi = \{ \text{this} \rightarrow \iota, x' \mapsto \chi', \gamma' \rightarrow \chi'(y) \}. \text{ By the induction hypothesis, there exist } \Sigma', \Gamma_2 \text{ such that if } H_1, \chi, s \sim H_2, \chi_1, \text{ then } \Sigma', \Gamma_2 \vdash H_2, \chi_2. \text{ By the induction hypothesis, } \forall \iota \in \text{dom}(\Sigma), \Sigma'(y) \leq \Sigma(y). \text{ Thus, } \forall \iota \in \text{dom}(\chi), \Sigma', \Gamma_0' \vdash \chi'(y), \forall \iota \in \text{dom}(\chi). \text{ Since } \Sigma'(y) \leq \Gamma'(x), \text{ we have } \Sigma', \Gamma_0' \vdash H_2, \chi_2(x \mapsto \chi_2') \text{ and } H_1 = H_1[\tau \mapsto \tau_2]. \text{ Thus, } S(V') = \text{def}(S(V') = \emptyset). \text{ Next we consider constraints on types.}

1. If \( m \leq V \leq m \in C \), then \( S(V) = \emptyset \).

2. If \( V \in \{ m : (V, \psi) \in C \} \), then by the definition of consistency, \( S(V, \psi) \) does not contain int or function types. Thus, \( \exists t, \text{ such that } S(V) = t \) or \( S(V, \psi) = \emptyset \). Thus, \( S(V, \psi) = \emptyset \). By Rule 4, \( V_2 = V_1 \) or \( V_2 \leq C, \) which implies \( S(V') = S(V_1) \).

3. If \( \forall V_0, M \times V_1 \leq V_2 \in C \), then by the definition of consistency, \( S(V, \psi) \) does not contain int or object types. Thus, \( \exists t, \text{ such that } S(V) = t \) or \( S(V, \psi) = \emptyset \). Thus, \( S(V, \psi) = \emptyset \). By Rule 5, \( V_0 \leq V_1 \) or \( V_2 \leq V_3 \), which implies \( S(V') = \emptyset \).

4. If \( V \leq V' \in C \), we consider the following subcases:
   (a) \( S(V') = \text{int} \) or \( S(V') = \emptyset \).
   (b) \( S(V') = \emptyset \).
   (c) \( S(V') = \emptyset \).
   (d) \( S(V) = t \) or \( S(V, \psi) = \emptyset \).

5. \( V \leq V' \in C \).

6. \( V \leq V' \in C \).

\textbf{Lemma D.1} \text{ If } E \vdash P : C' \text{ and } C = \text{Closure}(C') \text{ is consistent, then } \{ V \leq \Delta \} \text{ implies } V' \leq \{ m : \Delta \} \text{ in } C.

\textbf{Proof.} We prove by induction that if \( \{ V \leq \Delta, V', m \in M \} \leq C \) or \( V \leq \{ V', \Delta \} \times V_{\text{new}} \rightarrow V_{\text{reset}} \text{ in } C \), then \( V' \leq \{ m : \Delta \} \text{ is } \leq C \).

Let’s consider how constraints with \( M \) variables are added to \( C \) by the inference rules. For each member update, we generate constraints of the form \( V_{\text{new}} \leq \{ V_{\text{new}}, M \} \times V_{\text{new}} \rightarrow V_{\text{reset}} \text{ in } C \). For each method call, we generate constraints of the form \( V_g \leq \{ V_{\text{this}}, M \} \times V_{\text{new}} \rightarrow V_{\text{reset}} \text{ in } C \). For each function, we generate constraints of the form \( V_f \leq \{ V_{\text{this}}, M \} \times V_{\text{new}} \rightarrow V_{\text{reset}} \text{ in } C \).

\textbf{Lemma D.1} \text{ If } E \vdash P : C' \text{ and } C = \text{Closure}(C') \text{ is consistent, then } C \text{ is satisfiable.}

\textbf{Proof.} \text{ Since } C \text{ is Closed, it is } A\text{Closed as well.}

\textbf{Lemma 4.1} \text{ If } E \vdash P : C' \text{ and } C = \text{Closure}(C') \text{ is consistent, then } C \text{ is satisfiable.}

\textbf{Proof.} \text{ We show that } S = \text{Solution}(C) \text{ satisfies each constraint in } C.

First, \( S(M) = \{ m : m \in M \} \). \text{ S apparently satisfies constraint of the form } m \in M. \text{ For constraint of the form } M = \cup_{i \in I} M_i, \text{ since } C \text{ is Closed, by Rule 6, if } \{ m \in M_i \} \in C \text{ and } i \in I \text{, then } m \in M \text{ as well. Thus, } S(M) \subseteq S(M).

Since C is consistent, for each \( m \in M \in C \), \( \exists M_i \text{ such that } m \in M_i \text{ in } C \), \( \exists S(M) \subseteq S(M) \).

Therefore, \( S(M) \subseteq \cup_{i \in I} S(M_i) \). \text{ For constraint of the form } M = \cup_{i \in I} M_i, \text{ since } C \text{ is Closed, by Rule 6, if } \{ m \in M_i \} \subseteq C \text{ and } i \in I \text{, then } m \in M \text{ as well. Thus, } S(M) \subseteq S(M).

\textbf{Lemma 4.2} \text{ If } C \text{ is satisfiable, then } \text{Closure}(C) \text{ is consistent.}

\textbf{Proof.} \text{ We show that } S = \text{Solution}(C) \text{ is a solution to } C \text{ 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 \( U \subseteq V \), \( V \subseteq W \), then \( U \subseteq W \). By the definition of subtyping relation, it is transitive. Therefore, if \( S(U) \subseteq S(V) \) and \( S(V) \subseteq S(W) \), then \( S(U) \subseteq S(W) \).

**Rule 2** If \( \int \leq V \leq \tan \leq S(V) \) and \( S(V) = \int \leq \tan \).

**Rule 3** If \( V \leq \tan \leq S(V) \), then \( S(V) \leq \tan \leq S(V) \).

**Rule 4** If \( V \leq \tan \leq S(V) \), then \( S(V) \leq \tan \leq S(V) \).

**Rule 5** If \( V \leq (V_0, \mathcal{M}) \times V_2, V \leq (V_0', \mathcal{M}') \times V_2 \rightarrow V_2' \in \mathcal{C} \) and \( \exists \psi : (\mathcal{N}_\psi) \subseteq (\mathcal{N}_\psi) \), then \( S(V_1) = S(V_0) \). Since \( S(V_2) \leq S(V) \) and \( S(V) \leq S(V) \), by subtyping rules on object types, if \( S(V_1) = \tau \), then \( \tau \leq S(V) = S(V_2) \). Thus, \( S(V) \leq S(V_1) \) and \( S(V_2) \leq S(V) \).

**Rule 6** If \( m \in \mathcal{M}_1, j \in \{1, \ldots, n \}, M = \bigcup_{i=1}^n M_i, \tan \leq \mathcal{C} \), then \( (m, \mathcal{M}) \subseteq \bigcup_{i=1}^n (M_i, \mathcal{M}_i) \subseteq \mathcal{C} \). Since \( S \) is a solution to \( C \), we have \( m \in S((\mathcal{M}, \mathcal{M})) \) and \( S((\mathcal{M}, \mathcal{M})) \leq \mathcal{M} \). Thus, \( m \in \mathcal{M} \).

**Rule 7** If \( V \leq \tan \leq \mathcal{M}_1, \mathcal{M}_2 \subseteq \mathcal{C} \), then \( V \leq \mathcal{M}_1, \mathcal{M}_2 \subseteq \mathcal{C} \). Since \( \mathcal{M}_1 \leq \tan \leq \mathcal{M}_2 \), by definition of \( \mathcal{M} \), if \( m \notin \mathcal{M} \), then \( S(V) \) is possible for some \( \tau \) and \( \psi \), which means \( S(V) = \tau \). If \( m \in \mathcal{M} \), then \( S(V) = (\tau, \psi) \) for some \( \tau \) and \( \psi \). Therefore, \( S(V) \leq (m, S(V)) \).

**Rule 8** If \( V \leq \tan \leq \mathcal{M}_1, \mathcal{M}_2 \subseteq \mathcal{C} \), then \( V \leq \mathcal{M}_1, \mathcal{M}_2 \subseteq \mathcal{C} \). From \( S(V) \leq (m, S(V)) \), \( V \leq (m, S(V)) \), \( V \leq (m, S(V)) \).

**Rule 11** If \( V \leq \tan \leq \mathcal{M}_1, \mathcal{M}_2 \subseteq \mathcal{C} \), then \( V \leq (m, S(V)) \), \( V \leq (m, S(V)) \).

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

**Theorem 4.3** Given a program \( P \) where \( E \vdash P \mid C \), \( C \) is typable iff \( \text{Closure}(C) \) is consistent.

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

We first show that if \( C \) is satisfiable, then \( \exists \Sigma' \) such that \( \Gamma \vdash P \).

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

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

**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.

**Lemma E.1**. If \( \Sigma, \Gamma \vdash H(x, y) \) and \( \Gamma \vdash z : \tau \parallel \Gamma' \), then \( \exists \Sigma' \) such that

1. \( \Sigma', \Gamma' \vdash \chi(z) : \tau \)
2. \( \Sigma', \Gamma' \vdash H(x, y) \)
3. \( \forall t \in \text{dom}(\Sigma), \Sigma'(t) = \Sigma(t) \)

**Proof.** By the definition of \( \Gamma \vdash z : \tau \parallel \Gamma' \), either \( \Gamma(z) \leq \tau \) or \( \Gamma' = \Sigma' \) and \( \Sigma' = \Gamma \).

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

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

Finally, from \( \chi(z) \subseteq \Sigma', \chi''(z) \subseteq \Sigma \), we obtain the same set of members that are definite or labeled with *.

Thus, \( \Sigma', \Gamma' \vdash H'(z) : \chi''(z) \).

\[ \square \]
We now show that a well-typed program does not access undesigned object members.

**Lemma E.2.** If $\Sigma, \Gamma \vdash H, x : \tau \parallel \Gamma'$, then $H, \chi, x \vdash_{E} \tau$, and if $H, x : \tau \vdash_{E} \tau'$, then $H, \chi, x \vdash_{E} \tau'$. Consequently, if $C_1$ is the AClosure of $C'$ and $C_1 \rightarrow_C C_2$, then $C_2$ is also AClosed.

1. Suppose $V \not\subseteq (m, V')$. We need to show $S(V) \not\subseteq (m, S(V'))$. As explained above, $V$ is a vertex on a simple path in $G$. Also, by the type inference rules, $V$ may not be in another constraint of the form $V \not\subseteq V'$. Thus, $V' \not\subseteq V \subseteq C$ if $V \not\subseteq [m', V, o) \subseteq C$ for any $m' \neq m$.

Now consider three cases. The first case is when $m$ is not a member of $S(V')$. By Rule 13, $V \not\subseteq [m : (V, z), o) \subseteq C$. Then, $S(V') = (S(V'), z)$. The second case is when $S(V')(m) = (S(V'), z)$. The last case is when $S(V')(m) = (S(V'), \psi)$ where $\psi = \psi'$. By the definition of Solution, $C$ contains constraint of the form $V' \not\subseteq [m : (V', o), o]$ and Rule 14 adds $V \not\subseteq [m : (V', o), o]$ to $C$. In this case, $S(V)(m) = (S(V), \bullet)$. Therefore, $S(V) \not\subseteq (m, S(V), \bullet)$.

2. Suppose $V \subseteq V' \subseteq C$. If $S(V)(m) = (S(V), \psi)$, there exists a constraint of the form $V \not\subseteq [m : (V', \psi)]$. In Rule 15, $V \not\subseteq [m : (V', \psi)] \not\subseteq [m : (V', o), o] \subseteq C$ is in $C$. Thus, $S(V')(m) = (S(V'), \psi)$ and $\psi \not\subseteq \psi'$.

3. Suppose $V \not\subseteq V'$ in $C$. By Rule 16, if $V \not\subseteq [m : (V', o)]$, then $V' \not\subseteq [m : (V', o)] \subseteq C$. As explained above, $V'$ is a vertex on a simple path in $G$. By the inference rules, $V'$ may also appear in a constraint of the form $V' \not\subseteq V'$. But by the consistency rules, $V' \not\subseteq [m : (V, \psi)] \not\subseteq [m : (V', o), o] \subseteq C$. Therefore, $S(V')(m) = (S(V'), \psi)$, and $\psi \not\subseteq \psi'$.

4. Suppose $V \subseteq S(V') \subseteq C$. As explained above, $V$ is on a simple path in $G$. Also, by the inference rules, $V$ does not appear in a constraint of the form $V \not\subseteq V$. Moreover, the constraints of the form $V \not\subseteq (m, V, o)$ are added only by Rule 13, while the constraints of the form $V \not\subseteq [m : (V, o), o]$ are added only by Rule 15. Thus, by Rule 17, $V \not\subseteq [m : (V', o)]$ if $V' \not\subseteq [m : (V', o)] \subseteq C$ where $\psi = \psi'$ or $\psi = \psi'$. Similarly, by Lemma 1.1.1, we can show that if $m \in V_{m}'$, then $V' \not\subseteq [m : (V', o)] \subseteq C$. Hence, $\psi \not\subseteq \psi'$.

\[ \text{Lemma E.4. If } C \text{ is satisfiable, then } \text{Closure}(C) \text{ is consistent.} \]

**Proof.** We show that if $C$ 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 $C$, and $C \rightarrow_C C'$ or $C \rightarrow_C C'$, then $S$ is also a solution to $C'$. We will only consider the additional closure rules for the extended type system.

**Rule 12** If $\{V \in [m : (V', \psi)] : V \not\subseteq [m : (V', \psi)] \subseteq C\}$, then $\{V \not\subseteq V' \subseteq V \subseteq C\}$. Since $S$ solves $\{C(V) = (S(V))\}$, which implies $S(V') \subseteq S(V)$ and $S(V') \subseteq S(V)$.

**Rule 13** If $V \not\subseteq (m, V') \subseteq C$, then $V \not\subseteq [m : (V', o)]$. By definition, $S(V) \subseteq (m, S(V)) \subseteq (S(V), o)$ implies $S(V')(m) = (S(V), o)$, which implies $S(V')(m) = (S(V), o)$ and $S(V')(m) = (S(V), o)$, respectively.

**Rule 14** If $V \not\subseteq [m : (V', o)]$, then $V \not\subseteq [m : (V', o)] \subseteq C$ for any $m \neq m$ or $o = o$. Suppose $m \neq m$. Since $S(V) \subseteq (m, S(V)) \subseteq (S(V), o)$ and $S(V') \subseteq [m : (S(V'), o)]$, $S(V')(m) = (S(V'), o)$. 

\[\square\]
Suppose \( \psi = \emptyset \). Then, \( S(V)(m') = (S(V'), \psi') \) and \( \bullet \leq \psi' \).

Therefore, \( S(V) \leq [m' : (S(V'), \psi)] \).

**Rule 15** If \( V \leq V' \) and \( V \leq [m : (V', \psi)] \) are in \( C \), then \( V \leq [m : (V', \psi)] \) and \( V \leq [m : (V', \emptyset)] \) are in \( C' \). Since \( \delta(V) \leq S(V) \leq [m : (S(V'), \psi)] \), we have \( \psi \neq * \) and \( S(V) \leq [m : (S(V'), \psi)] \) and \( S(V) \leq [m : (S(V'), \emptyset)] \).

**Rule 16** If \( V \subseteq V' \) and \( V \leq [m : (V', \psi)] \) are in \( C \), then \( V' \leq [m : (V', \psi)] \) are in \( C' \). Since \( S(V) \subseteq S(V') \) and \( S(V)(m) = (S(V'), \psi) \), then \( S(V')(m) = (S(V'), \psi) \) and \( \psi_1 \leq \psi_2 \leq * \) or \( \psi_1 = \psi_2 = \emptyset \). Thus, \( S(V') \leq [m : (S(V'), \psi)] \).

**Rule 17** If \( V \leq_M V' \) and \( V' \leq [m : (V, \psi)] \) are in \( C \), then \( V \leq [m : (V, \psi)] \) in \( C' \). Since \( S(V) \leq_{S(M)} S(V') \), for \( m \notin S(M) \), \( S(V)(m) = S(V')(m) \), and for \( m \in S(M) \), \( S(V)(m) = (\tau, \bullet) \), \( S(V')(m) = (\tau, \psi) \), and \( \bullet \leq \psi' \). Thus, \( S(V) \leq [m : (S(V), \psi)] \).

**Rule 18** If \( V \leq_M V', m \in M \), and \( V' \leq [m : (V, \emptyset)] \) are in \( C \), then \( V \leq [m : (V, \bullet)] \) in \( C' \). Since \( S(V) \leq_{S(M)} S(V') \) and \( m \in S(M) \), \( S(V)(m) = (\tau, \bullet) \). Therefore, \( S(V) \leq [m : (S(V), \bullet)] \).

**Rule 19** If \( \mathcal{C} \rightarrow_A \mathcal{C} \), \( \{ V \leq_M V', V \leq [m : (V, \bullet)] \} \in C \), and \( \{ m \in M \} \notin C \), then \( V' \leq [m : (V, \bullet)] \) in \( C' \). Since \( S(V) \leq_{S(M)} S(V') \) and \( m \notin S(M) \), \( S(V)(m) = S(V')(m) \). Therefore, \( S(V) \leq [m : (S(V), \bullet)] \).

**Rule 20** If \( \mathcal{C} \rightarrow_A \mathcal{C} \), \( \{ V \leq_{M,V} V', V \leq [m' : (V', \bullet)] \} \in C \), then \( V' \leq [m' : (V', \bullet)] \) in \( C' \) where \( m' \neq m \).

Since \( S(V) \leq_{S(M)} S(V') \) and \( m' \neq m \), \( S(V)(m') = S(V')(m') \). Thus, \( S(V') \leq [m' : (S(V'), \bullet)] \).

**Rule 21** If \( \mathcal{C} \rightarrow_A \mathcal{C} \), \( \{ V \leq_{M} V', V \leq [m : (V, \bullet)] \} \in C \), then \( V \leq [m : (V, \bullet)] \) in \( C' \). Since \( S(V) \downarrow S(V') \) and \( S(V') \leq [m : (S(V), \bullet)] \), \( S(V)(m) = (S(V), \psi) \) and \( \psi \leq \bullet \). Thus, \( S(V) \leq [m : (S(V), \bullet)] \).