CHAPTER 3

Language Design and Implementation

3.1. Operational Semantics

In Section 2.1, we began the discussion of language specification and the importance specifying languages clearly, crisply, and precisely. Grammars is the main tool by which the syntax of a language, that is, the programs that we can write are specified. In this section, we introduce a tool for defining the semantics of a language, that is, the meaning of programs.

There are several ways to think about the meaning of program. One natural way is to think about how program evaluate. An operational semantics is a way to describe how programs evaluate in terms of the language itself (rather than by compilation to a machine model). One way to see an operational semantics is as describing an interpreter for the language of interest.

3.1.1. Syntax: JavaScripty. We consider a small subset of JavaScript, which we will affectionately call JavaScripty. The syntax of JavaScripty is given in Figure 3.1. Recall that we interpret such a definition as the abstract syntax of JavaScripty using elements from its concrete syntax. That is, we write concrete syntax for readability but assume that we are given abstract syntax trees that resolve ambiguity in the grammar.

\[
\begin{align*}
\text{expressions} & \quad e ::= x \mid n \mid b \mid \text{undefined} \mid \text{uop} e_1 \mid e_1 \text{ bop } e_2 \\
\text{values} & \quad v ::= n \mid b \mid \text{undefined} \\
\text{unary operators} & \quad \text{uop} ::= - \mid ! \\
\text{binary operators} & \quad \text{bop} ::= , \mid + \mid - \mid * \mid / \mid < \mid <= \mid > \mid >= \\
& \quad \mid == \mid != \mid && \mid || \\
\text{variables} & \quad x \\
\text{numbers (doubles)} & \quad n \\
\text{booleans} & \quad b ::= \text{true} \mid \text{false}
\end{align*}
\]

Figure 3.1. Syntax of JavaScripty
3. LANGUAGE DESIGN AND IMPLEMENTATION

Figure 3.1 describes JavaScript using a number of syntactic categories. The main syntactic category is expressions. We consider a program to be an expression. Expressions \( e \) consist of variables, value literals, unary operator expressions, binary operator expressions, a conditional if-then-else expression, a variable binding expression, and a print expression. Values \( v \) can be numbers (double-precision floating point), booleans, and a unique undefined value. This set of essentially arithmetic expressions is the usual core of any programming language. Functions are notably missing.

3.1.2. A Big-Step Operational Semantics. We might guess the semantics of particular expressions based on common conventions. For example, we probably guess that expression

\[ e_1 + e_2 \]

adds two numbers that result from evaluating \( e_1 \) and \( e_2 \). However, note that this statement something about the semantics of \( e_1 + e_2 \), which has yet to be specified.

One aspect that makes the JavaScript specification complex is the presence of implicit conversions (e.g., boolean values may be implicitly converted to numeric values depending on the context in which values are used). For example,

\[ \text{true} + 2 \]

evaluates to 3. How can we describe how to implement a JavaScript interpreter for all programs?

It is possible to specify the semantics of a programming language using natural language prose. However, just like with specifying syntax using natural language prose, it is very easy to leave ambiguity in the description. Furthermore, trying to minimize ambiguity can create very verbose descriptions. The JavaScript specification, specifically ECMA-262 standard [], is actually rather precise specification based on natural language prose, but the descriptions are quite verbose.

In this section, we introduce some mathematical notation that enables us to specify semantics with less ambiguity in a very compact form. Like any mathematical notation, its precise and compact nature makes it easier, for example, to spot errors or inconsistencies in specification. However, there will necessarily be a learning curve to reading the notation.

We want to write out as unambiguously as possible how a program should evaluate independent of an implementation (e.g., a compiler and
We use a method specification known as an operational semantics. An operational semantics can be thought as describing an interpreter for the language of interest with relations between syntactic objects. We have already used a notation for describing an evaluation relation:

\[ e \Downarrow v. \]

This notation is a judgment form stating informally, “Expression \( e \) evaluates to value \( v \).” Defining this judgment describes how to evaluate expressions to values and thus corresponds closely to writing a recursive interpreter of the abstract syntax trees representing expressions.

We will use a slightly richer judgment form with an additional parameter:

\[ E \vdash e \Downarrow v, \]

which says informally, “In value environment \( E \), expression \( e \) evaluates to value \( v \).” This relation has three parameters: \( E \), \( e \), and \( v \). The other parts of the judgment is simply punctuation that separates the parameters. The \( \vdash \) symbol is called the “turnstile” symbol.

A value environment \( E \) is a finite map from variables \( x \) to values \( v \) and can be described by the following grammar:

\[ E ::= \cdot | E[x \mapsto v]. \]

We write \( \cdot \) for the empty environment and \( E[x \mapsto v] \) as the environment that maps \( x \) to \( v \) but is otherwise the same as \( E \) (i.e., extends \( E \) with mapping \( x \) to \( v \)). Additionally, we write \( E(x) \) for looking up the value of \( x \) in environment \( E \). More precisely, we can define look up as follows:

\[
E[y \mapsto v](x) \overset{\text{def}}{=} v \quad \text{if } y = x \\
E[y \mapsto v](x) \overset{\text{def}}{=} E(x) \quad \text{otherwise} \\
\cdot(x) \overset{\text{def}}{=} \text{undefined}.
\]

The inference rules that define this evaluation judgment form is given in Figure 3.2. Let us first consider the two axioms:

\[ \text{EvalVar} \quad \text{EvalVal} \]

\[ E \vdash x \Downarrow E(x) \quad E \vdash \downarrow v \Downarrow v. \]

The EvalVar rule says that a variable use \( x \) evaluates to the value to which it is bound in the environment \( E \). Or operationally, to evaluate a variable use \( x \), look up the value corresponding to \( x \) in the environment \( E \). For an expression that is already a value \( v \), it evaluates to itself as stated by the EvalVal rule.

Now, back to the original example in this section, we are trying to specify how the expression

\[ e_1 + e_2 \]
evaluates. Thinking operationally, we want to say something like: evaluate \( e_1 \) to a number, evaluate \( e_2 \) to a number, and then return the number that is the addition of those numbers. Consider the following inference rule:

\[
\frac{E \downarrow e_1 \downarrow n_1 \quad E \downarrow e_2 \downarrow n_2 \quad n' = n_1 + n_2}{E \downarrow e_1 + e_2 \downarrow n'}
\]

Reading top-down, this rule says if we know that in environment \( E \), expression \( e_1 \) evaluates to a number \( n_1 \) and \( e_2 \) evaluates to \( n_2 \), then expression \( e_1 + e_2 \) evaluates to \( n' \) in environment \( E \) where \( n' \) is the addition of the \( n_1 \) and \( n_2 \). Note that the + in the premise is “plus” in the implementation language (i.e., the implementation language) in contrast to the + in the conclusion that is the syntactic symbol + in the object language (i.e., the source language). Here, we have highlighted the meta-language “plus” for clarity, but often, the reader is asked to determine this distinction based on context. To be completely explicit, we could use an alternative notation for the abstract syntax:

\[
\frac{E \downarrow e_1 \downarrow n_1 \quad E \downarrow e_2 \downarrow n_2 \quad n' = n_1 + n_2}{E \downarrow \text{Binary(Plus, } e_1, e_2 \text{) } \downarrow n'}
\]

This rule defines a semantics that does not fully match JavaScript because it requires \( e_1 \) and \( e_2 \) in \( e_1 + e_2 \) to evaluate to number values (i.e., \( n_1 \) and \( n_2 \)). JavaScript permits other types of values and then performs a conversion before performing the addition. We can express this semantics using the following inference rule:

\[
\frac{E \downarrow e_1 \downarrow v_1 \quad E \downarrow e_2 \downarrow v_2 \quad n' = \text{toNumber}(v_1) + \text{toNumber}(v_2)}{E \downarrow e_1 + e_2 \downarrow n'}
\]

Reading top-down, it says if we know that in environment \( E \), expression \( e_1 \) evaluates to \( v_1 \) and \( e_2 \) evaluates to \( v_2 \), then expression \( e_1 + e_2 \) evaluates to \( n' \) in environment \( E \) where \( n' \) is the addition of the toNumber conversions of \( v_1 \) and \( v_2 \). We define the toNumber conversion in Figure 3.3. This rule specifies the semantics that we want, though it is not the only way to do so.

Any evaluation rule can also be read bottom-up, which matches more closely to an implementation. For example, the above rule says, “To evaluate \( e_1 + e_2 \) in environment \( E \), evaluate \( e_1 \) in environment \( E \) to get value \( v_1 \), evaluate \( e_2 \) in environment \( E \) to get value \( v_2 \), convert \( v_1 \) and \( v_2 \) to numbers, and return the addition of those two numbers.”

In Figure 3.2, we lump all of the arithmetic operators +, −, *, and / together in the same rule: EvalArith. We use one notational shortcut
by treating the `bop` as the corresponding meta-language operator in the premise.

It is informative to study the complete set of inference rules and think about how the rules correspond to implementing an interpreter. The `EvalConst` rule is particularly interesting because we see explicitly that

$$\text{const } x = e_1; e_2$$

the scope of variable $x$ is the expression $e_2$ because $e_2$ is evaluated in an extended environment with a binding for $x$. 

**Figure 3.2.** Big-step operational semantics of JavaScript.
3. LANGUAGE DESIGN AND IMPLEMENTATION

### Figure 3.3. JAVASCRIPTY conversion functions.

<table>
<thead>
<tr>
<th>Function</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>toNumber(n)</td>
<td>def n</td>
</tr>
<tr>
<td>toNumber(true)</td>
<td>def 1</td>
</tr>
<tr>
<td>toNumber(false)</td>
<td>def 0</td>
</tr>
<tr>
<td>toNumber(undefined)</td>
<td>def NaN</td>
</tr>
<tr>
<td>toBoolean(n)</td>
<td>def false if n = 0 or n = NaN</td>
</tr>
<tr>
<td>toBoolean(true)</td>
<td>def true otherwise</td>
</tr>
<tr>
<td>toBoolean(b)</td>
<td>def b</td>
</tr>
<tr>
<td>toBoolean(undefined)</td>
<td>def false</td>
</tr>
</tbody>
</table>

3.2. Small-Step Operational Semantics

3.2.1. Evaluation Order. In Section 3.1.2, we have carefully specified several aspects of how the expression

\[ e_1 + e_2 \]

should be evaluated. In essence, it adds two integers that result from evaluating \( e_1 \) and \( e_2 \). However, there is still at least one more semantic question that we have not specified, “Is \( e_1 \) evaluated first and then \( e_2 \) or vice versa, or are they evaluated concurrently?”

Why does this question matter? Consider the expression:

\[ (\text{jsy.print}(1), 1) + (\text{jsy.print}(2), 2) \]

The operator is a sequencing operator. In particular, \( e_1, e_2 \) first evaluates \( e_1 \) to a value and then evaluates \( e_2 \) to value; the value of the whole expression is the value of \( e_2 \), while the value of \( e_1 \) is simply thrown away. Furthermore, `console.log(e_1)` evaluates its argument to a value and then prints to the screen a representation of that value. If the left operand of `+` is evaluated first before the right operand, then the above expression (3.2.1.1) prints 1 and then 2. If the operands of `+` are evaluated in the opposite order, then 2 is printed first followed by 1. Note that the final value is 3 regardless of the evaluation order.

The evaluation order matters because the `console.log(e_1)` expression has a side effect. It prints to the screen. As alluded to in Section 1.2, an expression free of side effects (i.e., is pure) has the advantage that the evaluation order cannot be observed (i.e., does not matter from the programmer’s perspective). Having this property is also known as being referentially transparent, that is, taking an expression and replacing any of its subexpressions by the subexpression's value cannot be observed as evaluating any differently than evaluating the expression itself. In JAVASCRIPTY, our only side-effecting expression is `console.log(e_1)`. If we remove the
prints from the above expression (3.2.1.1), then the evaluation order cannot be observed.

3.2.2. A Small-Step Operational Semantics of JAVASCRIPTY. The big-step operational semantics given in Section 3.1.2 does give us a nice specification for implementing an interpreter, but it does leave some semantic choices like evaluation order implicit. Intuitively, it specifies what the value of an expression should be (if it exists) but not precisely the steps to get to the value.

We have already used a notation for describing a one-step evaluation relation:

\[ e \rightarrow e' \]

This notation is a judgment form stating informally, “Expression \( e \) can take one step of evaluation to expression \( e' \).” Defining this judgment allows us to more precisely state how to take one step of evaluation, that is, how to make a single reduction step. Once we know how to reduce expressions, we can evaluate an expression \( e \) by repeatedly applying reduction until reaching a value. Thus, such a definition describes an operational semantics and intuitively an interpreter for expressions \( e \). This style of operational semantics where we specify reduction steps is called a small-step operational semantics.

In contrast to Section 3.1.2, we will not extend this judgment form with value environments. Instead, we define the one-step reduction relation on closed expressions, that is, expressions without any free variables. If we require the “top-level” program to be a closed expression, then we can ensure reduction only sees closed expressions by intuitively “applying the environment” eagerly via substitution. That is, variable uses are replaced by the values to which they are bound before reduction gets to them. As an example, we will define reduction so that the following judgment holds:

\[
\text{const } x = 1; \ x + x \rightarrow 1 + 1.
\]

This choice to use substitution instead of explicit environments is orthogonal to specifying the semantics using small-step or big-step (i.e., one could use substitution with big-step or environments with small-step). Explicit environments just get a bit more unwieldy here.

First, we need to describe what action does an operation perform. For example, we want to say that the + operator adds to numbers, which we say with the following rule:

\[
\text{DOPLUS} \quad \frac{n' = \text{toNumber}(v_1) + \text{toNumber}(v_2)}{v_1 + v_2 \rightarrow n'}
\]
This rules says the expression \( v_1 + v_2 \) reduces in one step to an integer value \( n' \) that is the addition of the toNumber conversion of values \( v_1 \) and \( v_2 \). We use the meta-variables \( v_1, v_2, \) and \( n' \) to express constraints that particular positions in the expressions must be values or numeric values. Note that the + in the conclusion is the syntactic + operator, while the + in the premise expresses mathematical addition of two numbers. As we discussed in Section 3.1.2, this symbol clash is rather unfortunate, but context usually allows us to determine which + is which. We sometimes call this kind of rule that performs an operation a *local reduction* rule. We will prefix all rules for this kind of rule with Do (and so will sometimes call them Do rules).

Second, we need to describe how we find the next operation to perform. These rules will capture issues like evaluation order described informally in Section 3.2.1. To specify that \( e_1 + e_2 \) should be evaluated left-to-right, we use the following two rules:

\[
\begin{align*}
\text{SearchPlus}_1 & \quad e_1 \rightarrow e'_1 \\
& \quad e_1 + e_2 \rightarrow e'_1 + e_2 \\
\text{SearchPlus}_2 & \quad e_2 \rightarrow e'_2 \\
& \quad v_1 + e_2 \rightarrow v_1 + e'_2
\end{align*}
\]

The *SearchPlus* \(_1\) rule states for an arbitrary expression of the form \( e_1 + e_2 \), if \( e_1 \) steps to \( e'_1 \), then the whole expression steps to \( e'_1 + e_2 \). We can view this rule as saying that we should look for an operation to perform somewhere in \( e_1 \). The rest of the expression \( \cdot + e_2 \) is a context that gets carried over untouched. The *SearchPlus* \(_2\) rule is similar except that it applies only if the left expression is a value (i.e., \( v_1 + e_2 \)). Together, these rules capture precisely a left-to-right evaluation order for an expression of the form \( e_1 + e_2 \) because (1) if \( e_1 \) is not a value, then only *SearchPlus* \(_1\) could possibly apply, and (2) if \( e_1 \) is a value, then only *SearchPlus* \(_2\) could possibly apply. We sometimes call this kind of rule that finds the next operation to perform a *global reduction* rule (or a *Search* rule). The sub-expression that is the next operation to perform is called the **redex**.

Considering these three rules, there is at most one rule that applies that specifies the “next” step. If our set of inference rules defining reduction has this property, then we say that our reduction system is *deterministic*. In other words, there is always at most one “next” step. Determinism is a property that we could prove about certain reduction systems, which we can state formally as follows:

**Property 3.1 (Determinism).** If \( e \rightarrow e' \) and \( e \rightarrow e'' \), then \( e' = e'' \).

In general, such a proof would proceed by structural induction on the derivation of the reduction step (i.e., \( e \rightarrow e' \)). We do not yet such proofs here in detail (cf., Section 2.3.3).
3.2. SMALL-STEP OPERATIONAL SEMANTICS

![Inference rules for small-step operational semantics](https://example.com/inference-rules.png)

**Figure 3.4.** Small-step operational semantics of JavaScript.

In Figure 3.4, we give all of the inference rules that define the one-step evaluation relation $e \rightarrow e'$ for JavaScript. The **DoNeg** states that the unary operator $-$ is integer negation, while **DoNot** states that $!$ is boolean negation. Observe that to “do” the operation, we require that the sub-expression under the unary operators $-$ or $!$, respectively, is a value. Contrast these rules to **EvalNeg** and **EvalNot** in Figure 3.2. If the sub-expression under the unary operator is not a value, then instead the rule **SearchUnary** applies telling us to look for something to reduce inside this sub-expression. The **DoSeq** rules states that the ‘,’ operator is used.

$$(x+y)[3/x] \rightarrow^* 3+y$$
def substitute(e: Exp, v: Exp, x: String): Exp:
  e match {
    case Var(y) =>
      if (x == y) v else Var(y)
    case ConstDecl(y, e1, e2) =>
      ConstDecl(y, substitute(e1, v, x),
                 if (x == y) e2 else substitute(e2, v, x))
  }
to indicate sequencing: for $e_1$, $e_2$, first $e_1$ is evaluated to a value, then that value is ignored, and we continue by evaluating $e_2$. The `DoArith`, `DoInequality`, and `DoEquality` specify how the arithmetic, inequality, and equality operators behave, respectively. The `DoArith` includes the case for + that we separated out in our discussion above.

We say that a short-circuit evaluation of expression is one where a value is produced before evaluating all subexpressions to values. The next four rules `DoAndTrue`, `DoAndFalse`, `DoOrTrue`, and `DoOrFalse` say that the boolean expressions $e_1 \&\& e_2$ and $e_1 || e_2$ may short-circuit. In particular, the rule

$$\text{DoAndFalse} \quad \begin{array}{l} \text{false} = \text{toBoolean}(v_1) \\ v_1 \&\& e_2 \rightarrow \text{false} \end{array}$$

says that $v_1 \&\& e_2$ where $v_1$ converts to `false` evaluates to `false` without ever evaluating $e_2$. The analogous rule for $||$ is

$$\text{DoOrTrue} \quad \begin{array}{l} \text{true} = \text{toBoolean}(v_1) \\ v_1 || e_2 \rightarrow \text{true} \end{array}$$

The `DoPrint` rule

$$\text{DoPrint} \quad \begin{array}{l} v_1 \text{ printed} \\ \text{console.log}(v_1) \rightarrow \text{undefined} \end{array}$$

is somewhat informal. In particular, since printing is outside of our model, the “$v_1$ printed” in the premise of the rule is not any required condition but should be viewed as comment for when this rule is applied. What is stated is the result of a `print` is the value `undefined`. For $e_1 ? e_2 : e_3$, the rules `DoIfTrue` and `DoIfFalse` specify with which expression to continue evaluation in the expected way depending on what boolean value to which the guard converts.

The `DoConst` rule for the variable binding expression `const x = e_1; e_2`

$$\text{DoConst} \quad \begin{array}{l} \text{const} \ x = v_1; e_2 \rightarrow e_2[v_1/x] \end{array}$$

is a bit more interesting. The expression-to-be-bound should already be a value $v_1$. We then proceed with $e_2$ with the value $v_1$ replacing the variable $x$. In general, the notation $e_1[e_2/x]$ is read as capture-avoiding substitution of expression $e_2$ for variable $x$ in $e_1$. We describe substitution in more detail below in Section 3.2.2.1.
The remaining rules in Figure 3.4 describe how to find the next operation to perform (i.e., the global reduction rules). They specify that all expressions are evaluated left-to-right.

3.2.2.1. **Substitution.** The term *capture-avoiding substitution* means that we get the expression that is like $e_1$, but we have replaced all instances of variable $x$ with $e_2$ while carefully respecting static scoping (cf., Section 1.2.4). There are two thorny issues that arise.

**Shadowing:** The substitution

\[
\begin{array}{c}
\text{(const } a = 1 ; a + b)[2 / a] \\
\text{e}_1
\end{array}
\begin{array}{c}
\text{e}_2
\end{array}
\begin{array}{c}
x
\end{array}
\]

should yield \((\text{const } a = 1 ; a + b)\). That is, only free instances of $a$ in $e_1$ should be replaced.

**Free Variable Capture:** The substitution

\[
\begin{array}{c}
\text{(const } a = 1 ; a + b)[(a + 2) / b] \\
\text{e}_1
\end{array}
\begin{array}{c}
\text{e}_2
\end{array}
\begin{array}{c}
x
\end{array}
\]

should yield something like \((\text{const } c = 1 ; c + (a + 2))\). In particular, the following result is wrong:

\[(\text{const } a = 1 ; a + (a + 2))\]

because the free variable $a$ in $e_2$ gets “captured” by the \textbf{const} binding of $a$.

In both cases, the issues could be resolved by renaming all *bound* variables in $e_1$ so that there are no name conflicts with free variables in $e_2$ or $x$. In other words, it is clear what to do if $e_1$ were instead

\[
\text{const } c = 1 ; c + b
\]

in which case textual substitution would suffice.

The observation is that renaming *bound* variables should preserve the meaning of the expression, that is, the following two expressions are somehow equivalent:

\[(\text{const } a = 1 ; a) \equiv_a (\text{const } b = 1 ; b)\]

For historical reasons, this equivalence is known $\alpha$-equivalence, and the process of renaming bound variables is called $\alpha$-renaming. This observation also leads to coming up with an abstract syntax representation so that the above two expressions are represented with the same object. As an aside, one way to do this is to use variables in the meta language to represent variables in the object language. This idea is known as *higher-order abstract syntax.*

In \textbf{DOCONST}, our situation is slight more restricted than the general case discussed above. In particular, the substitution is of the form $e[v/x]$
where the replacement for \( x \) has to be value. Values have no free variables, so only the shadowing issue arises.

In Figure 3.5, we define substitution \( e[e'/x] \) by induction over the structure of expression \( e \). As a pre-condition, we assume that \( e \) and \( e' \) use disjoint sets of bound variables. This pre-condition can always be satisfied by renaming bound variables appropriate as described above. Or if we require that \( e' \) has to be value, then this pre-condition is trivially satisfied. The most interesting cases are for variable uses and \texttt{const} bindings. For variable uses, we yield \( e' \) if the variable matches the variable being substituted for; otherwise, we leave the variable use unchanged. For a binding \texttt{const} \( x_1 = e_1; e_2 \), we recall that the scope of \( x_1 \) is \( e_2 \), so we substitute in \( e_2 \) depending on whether \( x_1 \) is \( x \). The remaining expression forms simply “pass through” the substitution.

3.2.2.2. Multi-Step Evaluation. We have now defined how to take one-step of evaluation. The multi-step evaluation judgment

\[ e \longrightarrow^{*} e' \]

says, “Expression \( e \) can step to expression \( e' \) in zero-or-more steps.” This judgment is defined using the following two rules:

\[
\begin{align*}
\text{\text{ZERO STEPS}} & \quad e \longrightarrow^{*} e \\
\text{\text{AT LEAST ONE STEP}} & \quad e \longrightarrow e' \quad e' \longrightarrow^{*} e'' \\
& \quad e \longrightarrow^{*} e''
\end{align*}
\]
3.3. Type Checking

In other words, $\rightarrow^*$ is the reflexive-transitive closure of $\rightarrow$.

A property that we want is that our big-step semantics and our small-step semantics are “the same.” We can state this property formally as follows.

\begin{property}[Big-Step and Small-Step Equivalence] $\vdash e \Downarrow v$ if and only if $e \rightarrow^* v$.
\end{property}

3.3. Type Checking

In Section 3.2, we defined a one-step reduction relation such that for any closed expression $e$: either $e$ is a value or $e \rightarrow e'$ for some $e'$, that is, $e$ can take a step to $e'$. This property is very nice but it came at a cost in complexity: we defined conversions between all types of values.

However, with a complex enough language, some types of values simply do not have sensible conversions. For example, let us consider extending JavaScript with function values. How should the number 3 convert to a function value?

In Figure 3.6, we extend JavaScript with first-class functions. The language of expressions are extended with function expressions $\text{function} \ (x) \ e_1$, which we consider shorthand for the following concrete syntax:

\begin{verbatim}
function (x) { return e1 }
\end{verbatim}

in JavaScript. For simplicity, we restrict functions to be anonymous and with exactly one argument. Function calls are written as $e_1(e_2)$. The language of values are also extend with function expressions, that is, function expressions are themselves considered values.

3.3.1. Getting Stuck. In Figure 3.7, we give additional rules for evaluating function calls. Observe in the \textsc{DoCall} rule that an evaluation step only makes sense if we are calling a function value. Otherwise, the set of rules simply say that call expressions are evaluated left-to-right and that both the function and the argument expressions must be values before continuing to evaluating with the body of the function. This latter choice is known as call-by-value semantics; we will return to this notion in ??.
3. LANGUAGE DESIGN AND IMPLEMENTATION

\[
\begin{align*}
\text{DoCall} & \quad (\text{function } (x) \ e_1)(v_2) \rightarrow e_1[v_2/x] \\
\text{SearchCall}_1 & \quad e_1 \rightarrow e'_1 \\
\text{SearchCall}_2 & \quad e_2 \rightarrow e'_2 \\
(\text{function } (x) \ e_1)(e_2) & \rightarrow (\text{function } (x) \ e_1)(e'_2)
\end{align*}
\]

**Figure 3.7.** Small-step operational semantics of JavaScript with first-class functions (extends Figure 3.4).

\[
\begin{align*}
\text{TypeErrorCall} & \quad v_1 \neq \text{function } p(x) \ e_1 \\
& \quad v_1(e_2) \rightarrow \text{typeerror} \\
\text{PropagateCall}_1 & \quad \text{typeerror}(e_2) \rightarrow \text{typeerror} \\
\text{PropagateCall}_2 & \quad v_1(\text{typeerror}) \rightarrow \text{typeerror}
\end{align*}
\]

**Figure 3.8.** Extending the small-step semantics of JavaScript from Figure 3.7 with dynamic type errors.

Note that these rules do not say anything about how to evaluate an ill-typed expression, such as

\[3(4)\]

Intuitively, evaluating this expression should result in an error. We do not state this error explicitly. Rather, we see that this an expression that is (1) not value and (2) can make no further progress (i.e., there's no rule that specifies a next expression). We call such an expression a *stuck expression*, which captures the idea that it is erroneous in some way.

### 3.3.2. Dynamic Typing

Another formalization and implementation choice would be to make such ill-typed expressions step to an error token. For example, we add to the expression language a token `typeerror` representing a dynamic type error:

\[
\text{expressions } \ e ::= \cdots | \text{typeerror}
\]

An ill-typed function call now steps to `typeerror` with rule `TypeErrorCall`. We also need to extend rules for evaluating other all other expression
forms that propagate the typeerror token if one is encountered in searching for a redex. We show PropagateCall_{1} and PropagateCall_{2}, which are two such rules, that correspond to SearchCall_{1} and SearchCall_{2}, respectively.

With this instrumentation, we distinguish a dynamic type error for any other reason for getting stuck. For example, an expression with free variables, such as

\[ x \]

Recall that our one-step evaluation relation is intended for closed expressions, so we might view this an internal error of the interpreter implementation rather than an error in the input JAVASCRIPTY program.

### 3.3.3. Static Typing

In Section 3.3.1, we saw how “bad” expressions, such as,

\[ 3(4) \]

are erroneous according to our operational semantics in that they “get stuck.” This expression gets stuck because a call expression \( e_{1}(e_{2}) \) only applicable to function values. We say that such an expression \( 3(4) \) is ill-typed or not well-typed.

A type is a classification of values that characterize the valid operations for these values. A type system consists of a language of types and a typing judgment that defines when an expression has a particular type. When we say that an expression \( e \) has a type \( \tau \), we mean that if \( e \) evaluates to a value, then that value should be of type \( \tau \). In this way, a type system predicts some property about how an expression evaluates at run-time.

In Figure 3.9, we show a language of types \( \tau \) for JAVASCRIPTY that includes base types for numbers \( n : \text{number} \), Booleans \( b : \text{bool} \), and the undefined value \( \text{undefined} : \text{Undefined} \), as well as a constructed type for function values. A function type \( (x : \tau) \Rightarrow \tau' \) classifies function values whose return value has type \( \tau' \) assuming its called with an argument of type \( \tau \). Our expression language \( e \) has modified slightly to add type annotations to function parameters.

A typing judgment form is defined by a set of typing rules that is the first step towards defining a type checking algorithm. A type error is an
expression that violates the prescribed typing rules (i.e., may produce a value outside the set of values that it is supposed to have). We define a typing judgment form inductively on the syntactic structure of program objects (e.g., expressions).

Recall from our earlier discussion on binding (Section 1.2.3) that the type of an expression with free variable depends on an environment. In other words, consider the expression

\[ x + 1 \]

Is this expression well-typed? It depends. If in the environment, \( x \) is stated to type \( \text{Int} \), then it is well-typed; otherwise, it is not. We see that the type of an expression \( e \) depends on a type environment \( \Gamma \) that gives the types of the free variables of \( e \). Thus, our typing judgment form is as follows:

\[
\Gamma \vdash e : \tau
\]

that says informally, “In typing environment \( \Gamma \), expression \( e \) has type \( \tau \).” Observe how similar this judgment form is to our big-step evaluation judgment form from Section 3.1.2. This observation is a bit more than a coincidence. A standard type checker works by inferring the type of an expression by recursively inferring the type of each sub-expression. A big-step interpreter computes the value of an expression by recursively computing the value of each sub-expression. In essence, we can view a type checker as an abstract evaluator over a type abstraction of concrete values.

In Figure 3.10, we define typing of JavaScript. The first four rules TYPENUMBER, TYPEBOOL, TYPEUNDEFINED, and TYPEFUNCTION describe the types of values. The types of the primitive values \( n \), \( b \), and \( \text{undefined} \) are as expected. The TYPEFUNCTION is more interesting:

\[
\frac{\Gamma \vdash e : \tau' \quad \Gamma[x \mapsto \tau] \vdash e : \tau}{\Gamma \vdash \text{function}(x : \tau) e : (x : \tau) \Rightarrow \tau'}
\]

A function value has a function type \((x : \tau) \Rightarrow \tau'\) (also sometimes called simply an “arrow” type) whose parameter type is \( \tau \) and return type is \( \tau' \). The return type \( \tau' \) is obtained by inferring the type of the body expression \( e \) under the extended environment \( \Gamma[x \mapsto \tau] \).

Examining the typing rules for the unary operators, we see that we have decided to restrict the input programs beyond those that get stuck with the small-step semantics defined in Figure 3.7:

\[
\text{TYPENEG} \quad \frac{\Gamma \vdash e_1 : \text{number}}{\Gamma \vdash \neg e_1 : \text{number}} \quad \text{TYPENOT} \quad \frac{\Gamma \vdash e_1 : \text{bool}}{\Gamma \vdash \neg e_1 : \text{bool}}
\]
### 3.3. Type Checking

For example, with rule `TypeNeg`, we say that $-e_1$ is well-typed if $e_1$ has type `number`. Furthermore, this rule is the only rule for $-e_1$, so we are only permitting unary negation – of numbers. Similarly, we are only permit not `!` applied to booleans. Thus, only code that does not need conversions is well typed. Therefore, we can simplify our interpreter to get rid
of conversions if only allow executing well-typed programs. We continue
with this design choice in this set of rules.

In the sequencing rule $\text{TypeSeq}$

$$
\frac{\Gamma \vdash e_1 : \tau_1 \quad \Gamma \vdash e_2 : \tau_2}{\Gamma \vdash e_1, e_2 : \tau_2},
$$

the type of the sequencing expression is $\tau_2$. The type of $e_1$ is checked but
then dropped—all we care about is that $e_1$ is well typed. Even though
we are dropping the type, the expression still needs to be type checked
because $e_1$ should evaluate to a value (without getting stuck) before being
dropped and continuing with the evaluation $e_2$.

The rule for conditionals $\text{TypeIf}$

$$
\frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \tau \quad \Gamma \vdash e_3 : \tau}{\Gamma \vdash e_1 ? e_2 : e_3 : \tau}
$$

looks a bit different than $\text{DoIfTrue}$ and $\text{DoIfFalse}$, the corresponding
big-step evaluation rules. In evaluation, we evaluate the guard expression
$e_1$ and continue with either $e_2$ or $e_3$ depending on whether $e_1$ evaluates
to $\text{true}$ or $\text{false}$. In type checking, we are predicting the type of values
that arise during execution before executing the program. This phase
is known as static-time or compile-time as opposed to dynamic-time or
run-time during execution.

In a sufficiently complex language (i.e., a Turing-complete language),
it is undecidable to precisely determine the value of an expression be-
fore executing it (i.e., statically), so we must approximate. Here, we re-
quire that both branches $e_2$ and $e_3$ have the same type $\tau$ because we
do not know whether $e_1$ will be $\text{true}$ or $\text{false}$ at run-time. This over-
approximation throws out some programs that would not get stuck at
run-time as a trade-off for being able to guarantee that well-typed pro-
grams do not get stuck. Over-approximation also happens in $\text{TypeCall}$.

This over-approximation requirement is the fundamental trade-off
between static and dynamic typing.