Translating JavaScript to **notJS**

1 Overall Translation Process

Our translation process consists of five phases that run in sequence. These are summarized below:

1. Parse in JavaScript using the parser from Mozilla's Rhino.

2. Convert the pure JavaScript AST from Rhino to our custom JavaScript-like $JSAST$.

3. Perform a series of $JSAST \rightarrow JSAST$ transformations on the resulting $JSAST$. These transformations are described in the "$JSAST$ to $JSAST$ Passes" section.

4. Convert the resulting $JSAST$ to a **notJS** AST using $\mathcal{T}$. This process, including the all-important $\mathcal{T}$ function, is described in the "JavaScript to **notJS**" section.

5. Perform a series of **notJS** $\rightarrow$ **notJS** passes on the **notJS** AST resulting from the previous phase. The only pass that is relevant to formalization is that of inserting **Merge** nodes in the appropriate places. This is described in the "**notJS** to **notJS** Passes" section.

1.1 Why $JSAST$ Exists

Based on the above summary, it may seem redundant that we introduce $JSAST$ as opposed to working directly with JavaScript ASTs from Rhino. There are four major reasons why this design decision was made, detailed below:

1. Rhino ASTs contain nodes that are atypical for ASTs. For example, there exists a special node in the Rhino AST for parenthesized expressions. From an abstract syntax standpoint, it is irrelevant whether or not a given expression was parenthesized in the concrete syntax; this sort of issue has already been handled during AST construction. Additional nodes merely complicate matters, especially in contexts when the translation needs to look at nested AST nodes.

2. It is useful to have some additional AST nodes available during translation which do not exist in pure JavaScript. These nodes are detailed in the "JavaScript Abstract Syntax" section. Adding these nodes to Rhino would require modifying the library itself, which seems undesireable from an engineering standpoint.

3. Certain APIs in Rhino behave in unexpected ways. For example, consider calls which return lists of items. For the vast majority of the APIs, if there are no items to return, then an empty list is returned. However, for certain APIs **null** is returned. Not only is this inconsistent, such behavior is not always well-documented, being instead revealed through experimentation. As such, it was desirable to develop a more consistent interface.

4. Rhino was written in Java, whereas our translation framework (and indeed our whole analysis) is written in Scala. While the two languages can coexist within the same instance of the Java Virtual Machine, the two languages have different idioms. For example, a common Java idiom is to use **null** as a sentinel value. However, such practice is strongly discouraged in Scala, with the preference being to use an `Option` type (i.e. the `Maybe` monad). If the Rhino AST were used directly, it would require breaking Scala idioms and good software engineering practices in Scala, and so it was decided to use a pure Scala interface instead.

Of potential interest is the fact that a previous invocation of the translator was written without introducing $JSAST$, operating directly on Rhino AST nodes instead. This very quickly became so bloated and complex as to be unmaintainable, prompting a total rewrite to the translator that is described herein this document.

2 Assumptions Made Regarding the Input JavaScript AST

The translator makes several assumptions about the input JavaScript AST. These assumptions must hold in order for translation to succeed and be correct. These assumptions are detailed below:

1. The source program does not use the JavaScript `with` construct. If `with` is required, then it is necessary to first apply the automated technique in Park et al. [2], which rewrites `with` in terms of non-`with`-containing JavaScript. Park et al.'s tool has been included in the supplementary materials.

2. No syntax or regular expression errors exist in the source program. Translation will terminate with an error for syntax errors, which comes for free from Rhino. However, for regular expression errors, Rhino misses certain invalid regular expressions. In this case, the translated code would fail at runtime.

3. No variables are used that have not first been declared, both at a function and global level. In other words, the source program will never throw a JavaScript `ReferenceError`. If this is violated, the translator can silently produce an incorrect translation.

4. The formal parameter list of any function does not contain repeated identifiers. Our translator will terminate with an error if this assumption does not hold.

5. `return` statements exist only within the body of a function. This comes for free from Rhino, which considers `return` statements at the global level syntax errors.

6. `break` statements exist only within the body of loops, the body of `switch` statements, and within statements with a label. Additionally, if `break` is provided with a label, the `break` must be lexically enclosed beneath the label. In other words, in JavaScript, only backward jumps are permitted. This is unlike C which permits code to jump both forwards and backwards between labels. If within a labeled statement, then the `break` must be provided a label that the `break` is lexically enclosed under. This sort of checking all comes for free from Rhino in our translator, which considers all violating cases syntax errors.

7. `continue` statements only exist within the body of a loop. As with `break`, if a label is provided, then the `continue` must be lexically closed underneath the provided label. Once again, this sort of checking comes for free from Rhino, which considers offending cases as syntax errors.

8. Assignments are made only to variables. Rhino also considers violations of this assumption as syntax errors.

Errors 2-8 are termed as "early errors" in the ECMA standard. We assume that input programs have been validated first by `JSLint` [1] or related tools, which can soundly detect these sort of errors as long as `eval` or its relatives (e.g., `setTimeout`) is not present.

3 Formalism Notation Explained

An arrow over something indicates that there is an ordered list of that something. For example, $\vec{x}$ indicates an ordered list of variables. Ordered lists can possibly be empty.

The standard inductive list definition is used. The empty list is indicated by **nil**, and all lists are terminated by **nil**. Individual elements can be prepended onto an existing list with cons (::). A list can be appended onto the front of another list with ++.

A bar over something indicates that there is a set of that thing. For example, $\bar{x}$ indicates a set of variables. As per the typical definition, sets can be empty.

It is possible to have a sequence of sets. For example, $\vec{\bar{x}}$ indicates a sequence of sets of variables.

As in **notJS**, $\mathcal{O}$ is used to denote an "Option" type, which we write as the polymorphic type $\mathcal{O}(A) = \textsf{none} + \textsf{some}\ A$. We denote variables having this type by o_x, where x is a destructor for any domain we have defined. For example, o_{str} represents an optional string, and $o_{\langle x,s \rangle}$ denotes an instance of an optional (variable, statement) pair.

Bindings can be both constructed and destructed like pairs. Indeed, notationally bindings are syntactic sugar for pairs. For example, in `var` $\overrightarrow{x = o_e}$, one can view the $\overrightarrow{x = o_e}$ portion as both a sequence of variable, optional expression bindings, or as a sequence of variable, optional expression pairs. This rule applies to `var` $\overrightarrow{x = o_e}$, `tempvar` $\overrightarrow{y = e}\ e'$, and also to $\left\{\overrightarrow{str \colon e}\right\}$.

For variables, generally y is used for temporary variables, and x is used for program-defined variables.

During the translation process, it is common for functions to return 3-tuples of $Stmt \times Exp \times \overline{Variable}$. Of special importance here is the recursive translation routine, described below:

$\mathcal{T}[\![\cdot]\!] \in JSAST \rightarrow (Stmt \times Exp \times \overline{Variable})$

Generally, the contract is that the statement returned must be executed in order to give meaning to the expression returned. The variables returned indicate whatever temporary variables were needed to be introduced in order to perform the translation. For example, say we have the code in the original JavaScript:

```
foo(5) + foo(6);
```

There does not exist a one-to-one mapping between this code and **notJS**, since in **notJS** function calls are statements instead of expressions. Instead, this translates to something like so:

```
var temp1 = foo(5);
var temp2 = foo(6);
temp1 + temp2;
```

At the heart of the above translation are the translations for $\texttt{foo}(5)$ and $\texttt{foo}(6)$. Such tranlations return 3-tuples similar to the following:

$$
\begin{aligned}
&\mathcal{T}[\![\texttt{foo}(5)]\!] = \\
&\quad (\texttt{temp1} = \texttt{foo}(5), \texttt{temp1}, \{\texttt{temp1}\}) \\
&\mathcal{T}[\![\texttt{foo}(6)]\!] = \\
&\quad (\texttt{temp2} = \texttt{foo}(6), \texttt{temp2}, \{\texttt{temp2}\}) \\
&\mathcal{T}[\![\texttt{foo}(5) + \texttt{foo}(6)]\!] = \\
&\quad (\texttt{temp1} = \texttt{foo}(5); \texttt{temp2} = \texttt{foo}(6) \\
&\quad\ \texttt{temp1} + \texttt{temp2}, \\
&\quad\ \{\texttt{temp1}, \texttt{temp2}\})
\end{aligned}
$$

Because this pattern is repeated so frequently, a type alias named *TranslationRetval* is used to more compactly represent $(Stmt \times Exp \times \overline{Variable})$.

4 JavaScript Abstract Syntax

$$n \in \mathit{JSNum} \qquad b \in \mathit{JSBool} \qquad \mathit{str} \in \mathit{JSStr} \qquad \ell \in \mathit{JSLabel} \qquad x, y \in \mathit{JSVar}$$

$$\begin{array}{rcl}
\mathit{jsast} \in \mathit{JSAST} & ::= & d \mid s \mid e \\
d \in \mathit{JSToplevelDecl} & ::= & \langle \overline{x}, s \rangle \\
s \in \mathit{JSStmt} & ::= & \vec{s} \mid e \mid \texttt{if}\ e\ s\ o_s \mid \mathit{loop} \mid \texttt{var}\ \overrightarrow{x = o_e} \mid \texttt{fun}\ \mathit{str}\ \vec{x}\ s \\
& & \mid \texttt{throw}\ e \mid \texttt{try}\ s\ \texttt{catch}\ o_{\langle x,s \rangle}\ \texttt{fin}\ o_s \mid \vec{\ell}\ s \mid \texttt{break}\ o_\ell \\
& & \mid \texttt{continue}\ o_\ell \mid \texttt{return}\ o_e \mid \texttt{switch}\ e\ \vec{w} \\
& & \mid \texttt{tempvar}\ \overrightarrow{y = e}\ e' \\
\mathit{case} \in \mathit{JSCase} & ::= & \texttt{switchCase}\ e\ s \\
w \in \mathit{JSSwitchSegment} & ::= & \mathit{case} \mid \texttt{switchDefault}\ s \\
\mathit{loop} \in \mathit{JSLoop} & ::= & \texttt{while}\ e\ s \mid \texttt{do}\ s\ \texttt{while}\ e \mid \texttt{for}\ s_1\ e\ s_2\ s_3 \mid \texttt{for}\ \mathit{lhs}\ \texttt{in}\ e\ s \\
e \in \mathit{JSExp} & ::= & n \mid b \mid \mathit{str} \mid \texttt{undef} \mid \texttt{null} \mid \mathit{lhs} \mid e_1 \oplus e_2 \mid \odot e \mid e_1\, ?\, e_2 : e_3 \\
& & \mid \mathit{lhs} = e \mid \mathit{lhs} \overset{\oplus}{=} e \mid e(\vec{e}) \mid \left\{ \overrightarrow{\mathit{str} : e} \right\} \mid [\vec{e}] \mid \texttt{regexp}\ \mathit{str}\ o_{\mathit{str}} \\
& & \mid \texttt{new}\ e(\vec{e}) \mid \texttt{fun}\ o_{\mathit{str}}\ \vec{x}\ s \mid \texttt{del}\ e \mid \texttt{this} \mid \varnothing \mid {+}{+}\mathit{lhs} \\
& & \mid \mathit{lhs}{+}{+} \mid {-}{-}\mathit{lhs} \mid \mathit{lhs}{-}{-} \\
\mathit{lhs} \in \mathit{JSLHS} & ::= & x \mid e_1[e_2] \\
\oplus \in \mathit{JSBop} & ::= & + \mid - \mid \times \mid \div \mid \% \mid \ll \mid \gg \mid \ggg \mid < \mid \leq \mid > \mid \geq \mid \& \mid '|' \\
& & \mid \veebar \mid \&\& \mid '||' \mid == \mid != \mid === \mid !== \mid , \mid \texttt{in} \mid \texttt{instanceOf} \\
\odot \in \mathit{JSUop} & ::= & - \mid \sim \mid \neg \mid + \mid \texttt{void} \mid \texttt{typeof} \mid \texttt{toObj}
\end{array}$$

Not all of the above nodes exist in standard JavaScript. The following three nodes are inserted during translation, and are specific to our own translation process:

1. Toplevel variable declarations ($\langle \overline{x}, s \rangle$). These define truly global variables - **not** properties of the `window` object as per the usual JavaScript definition of global variables. There is no direct analog in pure JavaScript.

2. Temporary variable declarations ($\texttt{tempvar}\ \overrightarrow{y = e}\ e'$). These are introduced in contexts where JavaScript expressions are needed, but temporary variables also need to be introduced within the same expression context. The intuitive semantics are that the given variable bindings are performed ($\overrightarrow{x = e}$), and then e' is evaluated and its value returned. This is needed because the standard JavaScript variable declaration ($\texttt{var}\ \overrightarrow{x = o_e}$) behaves only as a statement. This is a relatively short-lived AST node that is later stripped away.

3. Unary operations involving the `toObj` operator. The `toObj` operator converts the given argument to an object. Pure JavaScript has no directly equivalent operation.

A potential point of confusion is that many of the same metavariables used in the **notJS** syntax are reused for slightly different purposes in the *JSAST* definition, namely n, b, str, x, ℓ, s, e, d, $\oplus$, and $\odot$. This is exacerbated by the fact that these can be intermixed within the same portion of the formalism. For example, a given helper function may take a *JSStmt* and internally produce a *Stmt*, both of which use s as a metavariable for representation. It should be clear from context as to the underlying type of the metavariable.

5 Special Variables

There are several special variables that exist in any **notJS** program. `window` is one such special variable. Both a program-accessible and a program-inaccessible version of `window` exists. The program-accessible version of `window` is simply a variable named "window". The program-inaccessible version is used during translation; all subsequent uses of `window` refer to this special inaccessible version.

Another special variable is that of `self`. The `self` variable is automatically introduced as a hidden parameter to functions, and is used to dynamically determine what to bind JavaScript's `this`to. It is guaranteed that this variable will not conflict with program-defined variables.

Yet a third special variable is that of `arguments`. `arguments` is actually a JavaScript variable, one automatically in scope for function bodies. This variable encapsulates the arguments which have been passed to the function, and is often used for functions that take a variable number of arguments. It is the translator's responsibility to define the `arguments` variable for source programs.

In addition to `window`, `self`, and `arguments` there also exist the following other special variables, each given a value in the `preambleBindings` function:

- x_{dummy}: A dummy variable used in positions where a variable is required but its value is irrelevant. It is most frequently utilized to form a dummy statement of the form $x_{dummy} :=$ **undef**, used as a no-op in positions where a statement is required.
- x_{array}: A variable that always holds onto the global "Array" object.
- x_{regex}: A variable that always holds onto the global "RegExp" object.
- x_{number}: A variable that always holds onto the global "Number" object.
- $x_{arguments}$: A variable that always holds onto the global "Arguments" object.
- x_{object}: A variable that always holds onto the global "Object" object.
- $x_{dummyAddr}$: Exists to get around a cyclical need to pass an `Arguments` object to the constructor of `Arguments`. The concrete and abstract **notJS** interpreters recognize the address held within $x_{dummyAddr}$ as a special case specifically for `Arguments` for `Arguments` objects, breaking the cycle.

The above variables are inserted as necessary during translation.

None of the above variables are accessible by the user program. This is guaranteed by giving them names which are not valid JavaScript identifiers, and by putting them into the aforementioned toplevel variable declaration ($\langle \overline{x}, s \rangle$). Only through both mechanisms is it guaranteed that user programs cannot interfere with these variables. If they were valid identifiers, then the user program could trivially contain the same identifier. If they were properties of `window` (as with typical JavaScript global variables), then user code could still programatically access them, as with:

```
var internalName = "123Invalid+Identifier";
var hiddenVariable = window[internalName];
window[internalName] = null;
```

6 Special Labels

There are three special labels that are introduced during translation. These special labels are inaccessible from program code. These are detailed below:

- ℓ_{break}: Inserted just outside of contructs that allow for `break` statements to be inserted, namely loops and `switch`. Break statements are simply translated as a jump to ℓ_{break}.
- $\ell_{continue}$: Inserted at the head of loops to specifically handle `continue`. `continue` statements are translated as a jump to $\ell_{continue}$.
- ℓ_{return}: Inserted at the head of the body of functions in order to handle `return` statements. `return` statements are translated as a jump to ℓ_{return}.

It is guaranteed that user code cannot use these label names by using label names that are invalid in standard JavaScript, similar to the manipulation for translation-specific variables in the "Special Variables" section. Unlike with variables, no additional work is necessary, as label names cannot be computed with JavaScript. (The only exception to this is through JavaScript's `eval` construct.)

7 Helper Functions

We define the helper functions used throughout the translator. The functions are listed in alphabetical order.

7.1 `optionMap`

The `optionMap` helper is analogous to `map` for lists. It takes a function from A to A and applies it to the A contained within the option instance, if it exists. Otherwise, it simply returns **none**.

$$\texttt{optionMap} \in (A \to A) \times \mathcal{O}(A) \to \mathcal{O}(A)$$

$$\texttt{optionMap}(f, o) = \begin{cases} \textsf{\textbf{some}}\ f(a) & \text{if } o = \textsf{\textbf{some}}\ a \\ o & \text{otherwise} \end{cases}$$

7.1.1 `getOrElse`

The `getOrElse` helper is another function on instances of the option type that returns the object inside the option if it exists, or a provided alternative otherwise.

$$\texttt{getOrElse} \in \mathcal{O}(A) \times A \to A$$

$$\texttt{getOrElse}(a, o) = \begin{cases} a' & \text{if } o = \textsf{\textbf{some}}\ a' \\ a & \text{otherwise} \end{cases}$$

7.2 `call`

The `call` helper function generates the **notJS** code corresponding to a call of e_1 with e_2 as `self` and $\vec{e}$ as `args`, the result of which is stored in x.

$$\texttt{call} \in \mathit{Variable} \times \mathit{Exp} \times \mathit{Exp} \times \overrightarrow{\mathit{Exp}} \to \mathit{TranslationRetval}$$

$$\begin{aligned} &\texttt{call}(x, e_1, e_2, \vec{e}) = \\ &\quad \text{let } (s_{args}, e_{args}, \bar{y}) = \texttt{makeArguments}(\vec{e}) \\ &\quad (s_{args}; x := e_1(e_2, e_{args}), \\ &\quad\ \textsf{\textbf{undef}}, \bar{y}) \end{aligned}$$

7.3 `toSomething`

The role of `toSomething` is to generate the **notJS** code which will perform JavaScript's various implicit conversions explictly. It takes a **notJS** expression, together with a function that describes the conversion that is to be done if the expression is primitive and a function that explains what is to be done if the expression is not primitive. The non-primitive conversions requre a variable to perform scratch computations.

$$\texttt{toSomething} \in \mathit{Exp} \times (\mathit{Exp} \to \mathit{Exp}) \times (\mathit{Variable} \times \mathit{Exp} \to \mathit{TranslationRetval}) \to \mathit{TranslationRetval}$$

$$\begin{aligned} &\texttt{toSomething}(e, f_1, f_2) = \\ &\quad (\texttt{if } (\textsf{\textbf{isprim}}(e))\ \{y := f_1(e)\ \}\ \texttt{else}\ \{\ s\ \}, y, \bar{y} \cup \{y\}) \\ &\ \text{where} \\ &\quad (s, \_, \bar{y}) = f_2(y, e) \\ &\quad y \text{ is fresh} \end{aligned}$$

7.4 `toNumber`

The `toNumber` helper function is simply an instance of `toSomething` which relies on the helpers which are given below.

$$\texttt{toNumber} \in \mathit{Exp} \to \mathit{TranslationRetval}$$

$$\texttt{toNumber}(e) = \texttt{toSomething}(e, e \Rightarrow \textsf{\textbf{tonum}}(e), \texttt{callNumber})$$

7.5 toString

$\texttt{toString} \in \mathit{Exp} \rightarrow \mathit{TranslationRetval}$

$\texttt{toString}(e) = \texttt{toSomething}(e, e \Rightarrow \textsf{tostr}(e), \texttt{callToString})$

7.5.1 callNumber

This helper describes the **notJS** code that converts a non-primitive to a number, by calling the built-in `"Number"` function through x_{number}. This uses the program-inaccessible version of `window`.

$\texttt{callNumber} \in \mathit{Variable} \times \mathit{Exp} \rightarrow \mathit{TranslationRetval}$

$\texttt{callNumber}(x, e) = \texttt{call}(x, x_{number}, \texttt{window}, e :: \textsf{nil})$

7.6 callToString

The `callToString` helper is a stub that is used by other helpers to facilitate calling `toString` on an object.

$\texttt{callToString} \in \mathit{Variable} \times \mathit{Exp} \rightarrow \mathit{TranslationRetval}$

$\texttt{callToString}(x, e) = \texttt{callMethodByName}(x, e, \texttt{"toString"}, \textsf{nil})$

7.7 valueOf

The `valueOf` helper is used in making `"valueOf"` implicit conversions explicit.

$\texttt{valueOf} \in \mathit{Exp} \rightarrow \mathit{TranslationRetval}$

$\texttt{valueOf}(e) = \texttt{toSomething}(e, e \Rightarrow e, \texttt{callValueOf})$

7.7.1 callValueOf

This helper, like `callNumber`, describes how to explicitly convert a non-primitive to a primitive using `e."valueOf"`.

$\texttt{callValueOf} \in \mathit{Variable} \times \mathit{Exp} \rightarrow \mathit{TranslationRetval}$

$\texttt{callValueOf}(x, e) = \texttt{callMethodByName}(x, e, \texttt{"valueOf"}, \textsf{true} :: \textsf{nil})$

7.8 throwTypeError

This helper is simply a macro for throwing a type error.

$\texttt{throwTypeError} \in \mathit{Stmt}$

$\texttt{throwTypeError} = \textsf{throw}\ \text{“}\mathit{TypeError}\text{”}$

7.9 optionLabel

If the first label is defined, then it returns that label. Otherwise, it returns the other label.

$\texttt{optionLabel} \in \mathcal{O}(\mathit{JSLabel}) \times \mathit{JSLabel} \rightarrow \mathit{JSLabel}$

$\texttt{optionLabel}(o, \ell) = \texttt{getOrElse}(\ell, o)$

7.10 asNum

Converts the given integer to a **notJS** *Num*. This is considered basic, and so it has not been formalized.

$\texttt{asNum} \in \mathbb{Z} \rightarrow \mathit{Num}$

7.11 `prefixVarHelper`

Helper function for JavaScript expressions of the form $++\ x$ or $--\ x$. The integer parameter, i, specifies how much to increment or decrement by.

$\texttt{prefixVarHelper} \in \mathit{JSVar} \times \mathbb{Z} \rightarrow \mathit{TranslationRetval}$

$\texttt{prefixVarHelper}(x, i) =$

$\quad$ let $x' = \texttt{jsVarToNotJSVar}(x)$

$\quad$ let $(s_{num}, e_{num}, \bar{y}) = \texttt{toNumber}(x')$

$\quad$ $(s_{num}; x' := e_{num} + \texttt{asNum}(i),$

$\quad\;$ $x, \bar{y})$

7.12 `postfixVarHelper`

Helper function for JavaScript expressions of the form $x\ ++$ or $x\ --$. The integer parameter, i, specifies how much to increment or decrement by.

$\texttt{postfixVarHelper} \in \mathit{JSVar} \times \mathbb{Z} \rightarrow \mathit{TranslationRetval}$

$\texttt{postfixVarHelper}(x, i) =$

$\quad$ let $x' = \texttt{jsVarToNotJSVar}(x)$

$\quad$ let $(s_{num}, e_{num}, \bar{y}) = \texttt{toNumber}(x')$

$\quad$ let $y = y$ is fresh

$\quad$ $(s_{num};$

$\quad\;$ $y := e_{num};$

$\quad\;$ $x' := y + \texttt{asNum}(i),$

$\quad\;$ $y, \bar{y} \cup \{y\})$

7.13 `prepostAccessHelper`

A helper function for handling prefix/postfix increment/decrement on object access (i.e. $++\ e_1.e_2$). Takes four parameters:

1. An expression that should evaluate down to an object (i.e. e_1 in $e_1.e_2$).
2. An expression that should evaluate down to a field (i.e. e_2 in $e_1.e_2$).
3. A function that takes the result of $e_1.e_2$ as a guaranteed number and returns what the whole increment/decrement expression should return.
4. A function that takes the result of the previous function in a temporary variable. Returns an expression describing what the new value of $e_1.e_2$ should be.

$\texttt{prepostAccessHelper} \in \mathit{JSExp} \times \mathit{JSExp} \times (\mathit{Exp} \rightarrow \mathit{Exp}) \times (\mathit{Variable} \rightarrow \mathit{Exp}) \rightarrow \mathit{TranslationRetval}$

$\texttt{prepostAccessHelper}(e_1, e_2, f_1, f_2) =$

$\quad$ let $(s_{obj}, e_{obj}, \overline{y_{obj}}) = \mathcal{T}[\![e_1]\!]$

$\quad$ let $(s_{field}, e_{field}, \overline{y_{field}}) = \mathcal{T}[\![e_2]\!]$

$\quad$ let $(s_{access}, e'_{obj}, e'_{field}, \overline{y_{access}}) = \texttt{accessSetup}(e_{obj}, e_{field})$

$\quad$ let $(s_{num}, e_{num}, \overline{y_{num}}) = \texttt{toNumber}(e'_{obj}.e'_{field})$

$\quad$ let $y = y$ is fresh

$\quad$ $(s_{obj}; s_{field}; s_{access}; s_{num};$

$\quad\;$ $y := f_1(e_{num});$

$\quad\;$ $e'_{obj}.e'_{field} := f_2(y),$

$\quad\;$ $y, \overline{y_{obj}} \cup \overline{y_{field}} \cup \overline{y_{access}} \cup \overline{y_{num}} \cup \{y\})$

7.14 `prefixAccessHelper`

Helper for expressions of the form $++\ e_1.e_2$ or $--\ e_1.e_2$. The third parameter specifies how much to increment/decrement by.

$\texttt{prefixAccessHelper} \in \mathit{JSExp} \times \mathit{JSExp} \times \mathbb{Z} \rightarrow \mathit{TranslationRetval}$

$\texttt{prefixAccessHelper}(e_1, e_2, i) =$

$\quad \texttt{prepostAccessHelper}(e_1, e_2, e \Rightarrow e + \texttt{asNum}(i), y \Rightarrow y)$

7.15 `postfixAccessHelper`

Helper for expressions of the form $e_1.e_2\ ++$ or $e_1.e_2\ --$. The third parameter specifies how much to increment/decrement by.

$\texttt{postfixAccessHelper} \in \mathit{JSExp} \times \mathit{JSExp} \times \mathbb{Z} \rightarrow \mathit{TranslationRetval}$

$\texttt{postfixAccessHelper}(e_1, e_2, i) =$

$\quad \texttt{prepostAccessHelper}(e_1, e_2, e \Rightarrow e, y \Rightarrow y + \texttt{asNum}(i))$

7.16 `preambleBindings`

Returns a listing of bindings that are performed at the beginning of any **notJS** program. The fields `window."Arguments"` and `window."dummyAddress"` are specially provided to the translator, and do not exist in typical JavaScript.

$\texttt{preambleBindings} \in \overrightarrow{(\mathit{Variable} \times \mathit{Exp})}$

$\texttt{preambleBindings} =$

$\quad (x_{dummy}, \textbf{undef})\ ::$

$\quad (x_{array}, \texttt{window}.``Array")\ ::$

$\quad (x_{regex}, \texttt{window}.``RegExp")\ ::$

$\quad (x_{number}, \texttt{window}.``Number")\ ::$

$\quad (x_{arguments}, \texttt{window}.``Arguments")\ ::$

$\quad (x_{object}, \texttt{window}.``Object")\ ::$

$\quad (x_{dummyAddr}, \texttt{window}.``dummyAddress")\ ::\ \textbf{nil}$

7.17 `preamble`

Attaches a preamble to some translated program. The preamble sets up certain simple things in the environment, and is necessary for proper execution. The translator guarantees that the statement passed to `preamble`, namely s, will start with a variable declaration. The reassignments to the fields `window."Arguments"` and `window."dummyAddress"` are needed in order to prevent conflicts to the user program. Reassignment to **undef** instead of performing `delete` is sufficient. Since it is not possible to iterate over `window`, it is not possible for the user program to discover that the field exists but its value is **undef**.

$\texttt{preamble} \in \mathit{Stmt} \rightarrow \mathit{Stmt}$

$\texttt{preamble}(s) =$

$\quad \text{let } \textbf{decl}\ \overrightarrow{(x, e)}\ \textbf{in}\ s' = s$

$\quad \text{let } s_{final} =$

$\qquad \texttt{window}.``dummyAddress" = \textbf{undef}; \texttt{window}.``Arguments" = \textbf{undef}; s'$

$\quad \textbf{decl}\ (\overrightarrow{(x, e)}\ ++\ \texttt{preambleBindings})\ \textbf{in}\ s_{final}$

7.18 `translate`

Given a JavaScript AST that has undergone all the JavaScript → JavaScript transformations, translates it into a complete **notJS** AST.

$\mathtt{translate} \in \mathit{JSAST} \rightarrow \mathit{Stmt}$

$\mathtt{translate}(\mathit{jsast}) = \mathtt{preamble}(\mathtt{fst}(\mathcal{T}[\![\mathit{jsast}]\!]))$

7.19 `range`

Given two integers x and y, it will return all integers between x and y, inclusive, in increasing order.

$\mathtt{range} \in \mathbb{Z} \times \mathbb{Z} \rightarrow \overrightarrow{\mathbb{Z}}$

$$\mathtt{range}(i_1, i_2) = \begin{cases} \textbf{nil} & \text{if } i_1 > i_2 \\ i_1 :: \mathtt{range}(i_1 + 1, i_2) & \text{otherwise} \end{cases}$$

7.20 `jsVarToNotJSVar`

Converts a JavaScript variable into a **notJS** variable. This is considered a basic definition that is very implementation-specific, and so it has not been formalized.

$\mathtt{jsVarToNotJSVar} \in \mathit{JSVar} \rightarrow \mathit{Variable}$

7.21 `JSBopToBop`

Converts a JavaScript binary operator into a **notJS** binary operator. This is only defined for a subset of the JavaScript binary operators.

$\mathtt{JSBopToBop} \in \mathit{JSBop} \rightarrow \mathit{BinaryOp}$

$$\mathtt{JSBopToBop}(\oplus) = \begin{cases} - & \text{if } \oplus = - \\ \times & \text{if } \oplus = \times \\ \div & \text{if } \oplus = \div \\ \% & \text{if } \oplus = \% \\ \ll & \text{if } \oplus = \ll \\ \gg & \text{if } \oplus = \gg \\ \ggg & \text{if } \oplus = \ggg \\ \& & \text{if } \oplus = \& \\ | & \text{if } \oplus = | \\ \veebar & \text{if } \oplus = \veebar \end{cases}$$

7.22 `isTempVar`

Determines if a given JavaScript variable is a temporary one that was synthetically inserted during translation, or if the variable existed in the source program. This is considered a basic, implementation-specific predicate, and so it has not been formalized.

$\mathtt{isTempVar} \in \mathit{JSVar} \rightarrow \mathit{Bool}$

7.23 `intAsString`

Gets the string representation for an integer. For example, given the integer 5, the string representation would be “5”. This is considered a basic definition, and so it has not been formalized.

`intAsString` $\in \mathbb{Z} \rightarrow \mathit{String}$

7.24 `stringAsVar`

Lifts a string into the domain of variables. In other words, it treats a string as if it were a program variable of the same name. This is considered a basic definition, and so it has not been formalized.

`stringAsVar` $\in \mathit{String} \rightarrow \mathit{Variable}$

7.25 `varAsString`

The functional opposite to `stringAsVar`, though for JavaScript's *JSVar* instead of **notJS**' *Variable*. Given a JavaScript variable, it will return a string representation of its name. This is considered a basic definition, and so it has not been formalized.

`varAsString` $\in \mathit{JSVar} \rightarrow \mathit{String}$

7.26 `makeArguments`

Makes an `Arguments` object from the given expressions, which will be treated as individual parameters within the `Arguments` object.

$\texttt{makeArguments} \in \overrightarrow{\mathit{Exp}} \rightarrow \mathit{TranslationRetval}$

$\texttt{makeArguments}(\vec{e}) =$

let $y = y$ is fresh

let $i = \texttt{length}(\vec{e})$

$\texttt{addBinding} \in \mathit{Exp} \times \mathbb{Z} \rightarrow \mathit{Stmt}$

let $\texttt{addBinding}(e, i') =$

$y.\texttt{intAsString}(i') := e$

$(y := \texttt{new}\ x_{arguments}(x_{dummyAddr});$

$\texttt{asSeq}(\texttt{listMap}(\texttt{addBinding}, \texttt{zip}(\vec{e}, \texttt{range}(0, i-1))));$

$y.\text{“}\mathit{length}'' := i,$

$y, \{y\})$

7.27 `flatten`

Flattens a sequence of sets into a single set.

$\texttt{flatten} \in \vec{\bar{A}} \rightarrow \bar{A}$

$\texttt{flatten}(\vec{\bar{a}}) = \texttt{foldLeft}((\bar{a_1}, \bar{a_2}) \Rightarrow \bar{a_1} \cup \bar{a_2}, \emptyset, \vec{\bar{a}})$

7.28 `flattenVars`

Given a sequence of sets of variables, it flattens them into a single set of variables. Internally uses `flatten`. This is provided only so because this is the most common use of `flatten`, and it provides additional contextual information.

$\texttt{flattenVars} \in \overrightarrow{\overline{\mathit{Variable}}} \rightarrow \overline{\mathit{Variable}}$

$\texttt{flattenVars}(\vec{\bar{x}}) = \texttt{flatten}(\vec{\bar{x}})$

7.29 `asStmt`

Given a *JSAST*, it will return the *JSAST* as a *JSStmt*. This is only defined for inputs that are already *JSStmt*s; on any other input the result is undefined. Given that this is only a sort of typecast, it has not been formalized.

$\texttt{asStmt} \in \mathit{JSAST} \rightarrow \mathit{JSStmt}$

7.30 `asSeq`

Given a list of statements, it will treat them as a single statement. This is technically not required, as the grammar for **notJS** shows that a list of statements is a statement in and of itself. However, this is provided for clarity.

$\texttt{asSeq} \in \overrightarrow{\mathit{Stmt}} \rightarrow \mathit{Stmt}$

$\texttt{asSeq}(\vec{s}) = \vec{s}$

7.31 `asSeqJS`

Like `asSeq`, but applied to *JSStmt* instead of *Stmt*.

$\texttt{asSeqJS} \in \overrightarrow{\mathit{JSStmt}} \rightarrow \mathit{JSStmt}$

$\texttt{asSeqJS}(\vec{s}) = \vec{s}$

7.32 `toObj`

Ensures the given expression evaluates to an object via performing a conversion on it. If it is already an object, this conversion dynamically acts as a no-op.

$\texttt{toObj} \in \mathit{Exp} \rightarrow \mathit{TranslationRetval}$

$\texttt{toObj}(e) =$

let $y = y$ is fresh

$(y := \textbf{toobj}(e), y, \{y\})$

7.33 `accessSetup`

Helper function common to translations that involve accessing a particular field of a particular object. Given two expressions that should evaluate down to an object and a field name, respectively, this will return (in the following order):

1. A Statement that must be executed for the expressions that are returned to be valid.
2. An expression that is guaranteed to evaluate down to an object.
3. An expression that is guaranteed to evaluate down to a string.
4. Any temporary variables used during the process. These must be hoisted to their nearest enclosing scope.

$\texttt{accessSetup} \in \mathit{Exp} \times \mathit{Exp} \rightarrow \mathit{Stmt} \times \mathit{Exp} \times \mathit{Exp} \times \overrightarrow{\mathit{Variable}}$

$\texttt{accessSetup}(e_1, e_2) =$

let $(s_{obj}, e_{obj}, \overline{y_{obj}}) = \texttt{toObj}(e_1)$

let $(s_{field}, e_{field}, \overline{y_{field}}) = \texttt{toString}(e_2)$

$(s_{obj}; s_{field}, e_{obj}, e_{field}, \overline{y_{obj}} \cup \overline{y_{field}})$

7.34 `asList`

Gets the list representation of a set. The ordering of the resulting list is unspecified. This is considered a basic definition, so it has not been formalized.

$\texttt{asList} \in \bar{A} \rightarrow \vec{A}$

7.35 `asSet`

Gets the set representation of a list.

$\texttt{asSet} \in \vec{A} \rightarrow \bar{A}$

$\texttt{asSet}(\vec{a}) = \texttt{foldLeft}((\bar{a}, a) \Rightarrow \bar{a} \cup \{a\}, \emptyset, \vec{a})$

7.36 `noOpStatement`

A statement that is a no-op. Simply assigns **undef** to x_{dummy}.

$\texttt{noOpStatement} \in \mathit{Stmt}$

$\texttt{noOpStatement} = x_{dummy} := \textbf{undef}$

7.37 `noOpStatementExp`

Used to lift an expression to a *TranslationRetval*, using `noOpStatement` in the process. This is intended for pure expressions that need no statements to be evaluated for them to be meaningful, such as constants and variables.

$\texttt{noOpStatementExp} \in \mathit{Exp} \rightarrow \mathit{TranslationRetval}$

$\texttt{noOpStatementExp}(e) = (\texttt{noOpStatement}, e, \emptyset)$

7.38 `getBody`

Gets the body from a *JSSwitchSegment*, which is the statement within. Specifically, this means the code to execute if a given case is true, or the code within a `default` clause.

$\texttt{getBody} \in \mathit{JSSwitchSegment} \rightarrow \mathit{JSStmt}$

$$\texttt{getBody}(w) = \begin{cases} s & \text{if } w = \texttt{switchCase}\ e\ s \\ s & \text{if } w = \texttt{switchDefault}\ s \end{cases}$$

7.39 `whileHelper`

Helper to insert a `while` loop. This is needed to setup the ℓ_{break} and $\ell_{continue}$ labels correctly. Additionally, since `while`, `do/while`, and `for` loops all internally translate to `while` loops, this is needed to abstract away some commonality. It takes five parameters:

1. Some statement to execute after ℓ_{break} but before the loop is actually entered.
2. An expression that should evaluate to some *Bool* that the initial value of the guard is set to. It is safe if the given expression is not guaranteed to evaluate down to a *Bool* directly.
3. An expression that should evaluate down to whatever the guard should be after an iteration of the loop. It is safe if the given expression is not guaranteed to evaluate down to a *Bool* directly.
4. The body of the loop. $\ell_{continue}$ will automatically be inserted at the head of the body.
5. A statement to execute after the body of the loop has been executed.

$\texttt{whileHelper} \in \mathit{Stmt} \times \mathit{Exp} \times \mathit{Exp} \times \mathit{Stmt} \times \mathit{Stmt} \to \mathit{TranslationRetval}$

$\texttt{whileHelper}(s_1, e_1, e_2, s_2, s_3) =$

let $y = y$ is fresh

$(\ell_{break}:$

$s_1;$

$y := \textsf{\textbf{tobool}}(e_1);$

$\texttt{while}\ (y)\ \{$

$\ell_{continue}:$

$s_2;$

$s_3;$

$y := \textsf{\textbf{tobool}}(e_2)$

$\},$

$\textsf{\textbf{undef}}, \{y\})$

7.40 List Operations

7.40.1 `unzip`

Given a list of pairs, it will return a pair of lists respecting the same order as in the original list.

$\texttt{unzip} \in \overrightarrow{(A \times B)} \to \vec{A} \times \vec{B}$

$\texttt{unzip}(\overrightarrow{(a, b)}) = (\texttt{listMap}(\texttt{fst}, \overrightarrow{(a, b)}), \texttt{listMap}(\texttt{snd}, \overrightarrow{(a, b)}))$

7.40.2 `unzip3`

Given a list of 3-tuples, it will return a 3-tuple of lists respecting the same order as in the original list. This is in the same vein as `unzip`, but applied to 3-tuples. Due to the similarity with `unzip`, it has not been formalized.

$\texttt{unzip3} \in \overrightarrow{(A \times B \times C)} \to \vec{A} \times \vec{B} \times \vec{C}$

7.40.3 `unzip4`

Given a list of 4-tuples, it will return a 4-tuple of lists respecting the same order as in the original list. This is in the same vein as `unzip`, but applied to 4-tuples. Due to the similarity to `unzip`, it has not been formalized.

$\texttt{unzip4} \in \overrightarrow{(A \times B \times C \times D)} \to \vec{A} \times \vec{B} \times \vec{C} \times \vec{D}$

7.40.4 `zip`

Given two lists, it will return a list of pairs, where the first element of each pair is from the first list and the second element of each pair is from the second list.

$\texttt{zip} \in \vec{A} \times \vec{B} \to \overrightarrow{(A \times B)}$

$$\texttt{zip}(\vec{a}, \vec{b}) = \begin{cases} \textsf{\textbf{nil}} & \text{if } \texttt{isEmpty}(\vec{a}) \vee \texttt{isEmpty}(\vec{b}) \\ (\texttt{head}(\vec{a}), \texttt{head}(\vec{b})) :: \texttt{zip}(\texttt{tail}(\vec{a}), \texttt{tail}(\vec{b})) & \text{otherwise} \end{cases}$$

7.40.5 `foldLeft`

The typical definition of `foldLeft` over lists. This is considered a basic definition, so it has not been formalized.

$\texttt{foldLeft} \in ((B \times A) \to B) \times B \times \vec{A} \to B$

7.40.6 `foldRight`

The typical definition of `foldRight` over lists. This is considered a basic definition, so it has not been formalized.

$\texttt{foldRight} \in ((A \times B) \to B) \times B \times \vec{A} \to B$

7.40.7 `listMap`

The standard definition for `map` over lists.

$\texttt{listMap} \in (A \to B) \times \vec{A} \to \vec{B}$

$\texttt{listMap}(f, \vec{a}) = \texttt{foldRight}((a, \vec{b}) \Rightarrow f(a) :: \vec{b}, \textsf{nil}, \vec{a})$

7.40.8 `filter`

Returns a new list holding all the elements in the original list that match a given predicate.

$\texttt{filter} \in (A \to \mathit{Bool}) \times \vec{A} \to \vec{A}$

$\texttt{filter}(f, \vec{a}) =$

$$\text{let } \texttt{step}(a, \vec{a'}) = \begin{cases} a :: \vec{a'} & \text{if } f(a) \\ \vec{a'} & \text{otherwise} \end{cases}$$

$$\texttt{foldRight}(\texttt{step}, \textsf{nil}, \vec{a})$$

7.40.9 `partition`

Given a predicate and a list of inputs, returns a pair of lists holding which inputs matched and did not match the given predicate, respectively. For simplicity, this is implemented in terms of `filter`, which is safely possible to do due to mathematical purity.

$\texttt{partition} \in (A \to \mathit{Bool}) \times \vec{A} \to \vec{A} \times \vec{A}$

$\texttt{partition}(f, \vec{a}) =$

$$(\texttt{filter}(f, \vec{a}), \texttt{filter}(a \Rightarrow \neg f(a), \vec{a}))$$

7.40.10 `listMapOption`

Given a list of $\mathcal{O}(A)$ and some function $f \in A \to B$, it returns a new list of B, where each element results from applying f from a **some** element of $\overrightarrow{\mathcal{O}(A)}$. Note that the list that results from the internal call to `filter` only contains **some** elements; no **none** elements.

$\texttt{listMapOption} \in (A \to B) \times \overrightarrow{\mathcal{O}(A)} \to \vec{B}$

$\texttt{listMapOption}(f, \vec{o_a}) =$

$$\texttt{listMap}((\textsf{some } a) \Rightarrow f(a), \texttt{filter}(o_a \Rightarrow o_a \neq \textsf{none}, \vec{o_a}))$$

7.40.11 `head`

Gets the head element of a list. If the list is empty, `head` is undefined. This is considered basic, so it has not been formalized.

$\texttt{head} \in \vec{A} \to A$

7.40.12 `tail`

Gets a copy of the given list without the first element. If the list is empty, `tail` is undefined. This is considered basic, and so it has not been formalized.

$\texttt{tail} \in \vec{A} \rightarrow \vec{A}$

7.40.13 `length`

Gets the number of non-**nil** elements in the given list. This is considered basic, and so it has not been formalized.

$\texttt{length} \in \vec{A} \rightarrow \mathbb{Z}$

7.40.14 `last`

Gets the last element from the given list. This is undefined on an empty list.

$\texttt{last} \in \vec{A} \rightarrow A$

$\texttt{last}(\vec{a}) = \texttt{head}(\texttt{reverse}(\vec{a}))$

7.40.15 `isEmpty`

Determines whether or not the given list is empty. Following from the inductive list definition we are using, this is simply a check for **nil**.

$\texttt{isEmpty} \in \vec{A} \rightarrow Bool$

$\texttt{isEmpty}(\vec{a}) =$

$\vec{a} \stackrel{?}{=} \textbf{nil}$

7.40.16 `reverse`

Given a list, returns a new list in the opposite order of the input list.

$\texttt{reverse} \in \vec{A} \rightarrow \vec{A}$

$\texttt{reverse}(\vec{a}) = \texttt{foldLeft}((\vec{a'}, a) \Rightarrow a :: \vec{a'}, \textbf{nil}, \vec{a})$

7.41 Tuple Operations

7.41.1 `fst`

Gets the first element in an n-tuple, where $n >= 1$. This is considered basic, and so it has not been formalized.

$\texttt{fst} \in (A \times ...) \rightarrow A$

7.41.2 `snd`

Gets the second element in an n-tuple, where $n >= 2$. This is considered basic, and so it has not been formalized.

$\texttt{snd} \in (A \times B \times ...) \rightarrow B$

8 *JSAST* to *JSAST* Passes

8.1 Pass Descriptions

Before the main transformation pass runs that converts JavaScript into **notJS**, a series of passes come before that perform successive transformations on pure JavaScript. Many JavaScript forms can be more easily represented in JavaScript itself via desugaring. More accurrately, many *JSAST* forms can be more easily represented via desugaring of complex *JSAST* forms into simpler *JSAST* forms.

A listing of the different passes in the order in which they are performed follows. Variables with the prefix `temp` are automatically introduced, and exist in a namespace that is guaranteed separate from programmer-defined variable names. As mentioned before, references to `window` are actually to a special variable that is inaccessible to the JavaScript program.

1. **Replace Empty With Undef:** This replaces all uses of the empty expression (∅) with `undef`. This is simply to reduce the number of AST nodes that the JavaScript to **notJS** pass must handle.

2. **Fix Continue Labels:** For loops with user-defined labels, this will insert a special label at the head of the body of the loop to where `continue` will jump to. As an example, consider the following JavaScript code:

   ```
   user_label:
     while (x < 5) {
       if (x == 2) continue user_label;
       alert(x);
     }
   ```

   With a standard translation, `user_label` ends up being placed outside of the loop. This is perfectly fine for `break`, since a jump to `user_label` will force the loop to be escaped in that case. However, this is problematic for `continue`, since `continue` should only skip over the body of a loop for a single iteration. (In JavaScript, even for `continue`, the label must be placed immediately outside of the loop.) With this in mind, we transform the above code to the following:

   ```
   user_label:
     while (x < 5) {
       continue_user_label:
         if (x == 2) continue continue_user_label;
         alert(x);
     }
   ```

   That is, we insert a modified version of the same label at the head of the loop, and modify all `continue` statements to jump to the modified label. A new label is introduced to allow for `break` and `continue` to coexist in the same loop without changing the program semantics.

3. **Make All Assignments Simple:** JavaScript allows for assignments to be optionally annotated with certain binary operations, as in `x += y`. This pass will remove all such annotated assignments, replacing them with equivalent unanotated assignments which perform binary operations on the righthand side. Consider the following example:

   ```
   function field() {
     return "bar";
   }
   function foo(obj, y) {
     var x = 1;
     x += y;
     obj[field()] += y;
   }
   ```

   This pass will transform the above code into the code below:

   ```
   function field() {
     return "bar";
   ```

```
}
function foo(obj, y) {
  var temp1, temp2;
  var x = 1;
  x = x + y;
  temp1 = toObj(obj);
  temp2 = field();
  temp1.temp2 = temp1.temp2 + y;
}
```

In the above example, it may seem strange that `temp2` is repeated. After all, `temp2` could refer to an object with a `toString` method defined. This is significant, as it means that at runtime `toString` would be executed twice for a single update. While this may go against intuition, this is actually correct behavior according to ECMA.

4. **Hoist Functions:** Hoists function definitions to the top of their enclosing function or global scope. For example, consider the following JavaScript code (whole program):

```
function first() {}
alert("foo");
function second() {
  function nested1() {}
  alert("bar");
  var x = function nested2() {
    alert("baz");
    function nested3() {}
  };
  function nested4() {}
}
alert("boo");
```

This transforms into the code below:

```
function first() {}
function second() {
  function nested1() {}
  function nested4() {}
  alert("bar");
  var x = function nested2() {
    function nested3() {}
    alert("baz");
  };
}
alert("foo");
alert("boo");
```

5. **Hoist Variable Declarations:** Hoists variable declarations to the top of their enclosing scope. Also puts function names in the same namespace as variables, hoisting those variable declarations, and making them global properties of `window` as necessary. For example, consider the following JavaScript code (whole program):

```
function foo() {
  function bar() {}
  var x = 12;
  x++;
  var y = x + 5;
  var baz = function func() {
    var a = 7;
  }
}
```

This code translates to:

```
window.foo = undefined;
function foo() {
  var bar = undefined;
  var x = undefined;
  var y = undefined;
  var baz = undefined;
  var temp = undefined;
  function bar() {}
  x = 12;
  x++;
  y = x + 5;
  baz = temp = function func() {
    var func = temp;
    var a = undefined;
    a = 7;
  }
}
```

The variable `baz` is intentionally only assigned `undefined` in the above code, instead of its corresponding function. This is addressed in the later **Function Declaration to Expression** pass. This is deferred only to simplify the current pass (**Hoist Variable Declarations**).

6. **Make Global Variables `window` Properties:** For all variables that are not in scope, it makes them properties of the special predefined `window` object. This fully exploits the assumption that there are no reference errors in the code. If there were reference errors, this would erroneously make these properties of `window`. To illustrate this pass, consider the following JavaScript code below (whole program):

```
window.foo = undefined;
function foo(param) {
  var z = undefined;
  x = 12;
  y = x + param;
  try {
    z = 13;
  } catch (exc) {
    z = exc;
  }
}
```

This code translates to the following:

```
window.foo = undefined;
function foo(param) {
  var z = undefined;
  window.x = 12;
  window.y = window.x + param;
  try {
    z = 13;
  } catch (exc) {
    z = exc;
  }
}
```

7. **Function Declaration to Expression:** Converts all function declarations into equivalent function expressions. This is to simplify downstream translations, which then only need to consider function expressions instead of both declarations and expressions. For example, consider the following JavaScript code (whole program):

```
window.foo = undefined;
window.bar = undefined;
function foo() {}
function bar() {
  var baz = undefined;
  function baz() {}
}
```

This code translates to the following:

```
window.foo = undefined;
window.bar = undefined;
window.foo = function foo() {};
window.bar = function bar() {
  var baz = undefined;
  baz = function baz() {};
}
```

This pass takes advantage of the fact that function declaration names were already hoisted by **Hoist Variable Declarations**. It may appear that conflicts are introduced, since for each function declaration, the name of the declaration is the same as its corresponding variable name. However, this is not the case, as function names are simply strings in our definition. Function names are stripped away entirely in the translation to **notJS**.

8. **Remove `this`:** Replaces all uses of JavaScript's `this` with a variable. At the global scope, `this` refers to the `window` object. As such, at the global scope, this pass replaces `this` with `window`. Within a function scope, `this` refers to different things depending upon how the function was invoked. **notJS** handles `this` in functions by passing a hidden `self` parameter to functions, where `self` always refers to the correct reference for `this`. As such, within functions, we simply replace `this` with `self`. For example, consider the following JavaScript code (whole program):

```
window.x = undefined;
window.foo = undefined;
window.x = this;
window.foo = function foo() {
  var y = undefined;
  y = this;
}
```

This example translates to the following:

```
window.x = undefined;
window.foo = undefined;
window.x = window;
window.foo = function foo() {
  var y = undefined;
  y = self;
}
```

9. **Handle Catch Scoping:** In **notJS**, the variable introduced by the `catch` block in a `try`/`catch` statement must already be in scope. This pass will insert a new variable that is in scope for a given `try`/`catch` statement, and rename all uses of the original variable to the new variable within the corresponding `catch` block. For example, consider the following JavaScript code (whole program):

```
window.foo = undefined;
try {
  alert("a");
} catch (x) {
  alert(x);
```

```
}
window.foo = function foo() {
  try {
    alert("b");
  } catch (y) {
    alert(y);
  }
}
```

This code translates to the following:

```
var temp1 = undefined;
window.foo = undefined;
try {
  alert("a");
} catch (temp1) {
  alert(temp1);
}
window.foo = function foo() {
  var temp2 = undefined;
  try {
    alert("b");
  } catch (temp2) {
    alert(temp2);
  }
}
```

With the above example, the variable `temp1` is introduced as a truly global variable, one placed in the special *JSToplevelDecl*. This is in contrast to making `temp1` a property of `window`, which potentially could conflict with the source program's global variables.

8.2 Pass Framework

8.2.1 Intuition

All passes share a common structure. Each pass selectively replaces certain AST nodes with other AST nodes, depending on what exactly the pass is intended to do. For example, the **Replace Empty With Undef** pass replaces ∅ AST nodes with `undef` AST nodes.

Certain passes also need to build up and pass some sort of context down the AST. For example, the **Make Global Variables `window` Properties** pass needs to keep track of which variables are in scope. In order to do this, it must keep track of which variables have been declared, which means looking at variable declaration AST nodes. When a variable is later encountered, this information from variable declaration nodes can be used to determine whether or not the given variable is in scope.

Certain passes also need to forward information up the AST. For example, the **Hoist Functions** pass needs to forward function declarations up to the nearest enclosing scope.

Individual passes are generally concerned only with a small fraction of all the AST nodes. For example, the **Replace Empty With Undef** is only concerned with ∅ AST nodes. For this reason, the passes are implemented using partial functions which are defined only on AST nodes of interest. For nodes that do not exist in the domain of these functions, a default action is performed. This default action simply duplicates the underlying AST node, recursively applies the translation process, and and coalesces any information that is passed upward. By having default actions, the formalization of individual passes is much shorter, and free of large sections of duplication.

A common theme for passes is to manipulate something at either the function scope or the global scope. For example, the **Hoist Variable Declarations** pass will put variable declarations at the top of a function, and it will also set certain global properties of `window` to **`undef`**. It is possible to get a handle on the function scope by using partial functions defined on function declarations and expressions. It is possible to get a handle on the global scope in a similar fashion by exploiting *JSToplevelDecl*, though *JSToplevelDecl* does not exist in all passes. As such, there is a post-pass that is done: once the whole AST has been traversed, it is sent to a finishing function. It is guaranteed that the root of the AST is at the global scope, so this function is directly given a handle on the global scope.

8.2.2 Formalization

Passes are all implemented through the helper function `makePass`, which has the following type signature:

$$\texttt{makePass} \in D \times U \times ((U \times U) \to U) \times ((\mathit{JSAST} \times U) \to \mathit{JSAST}) \times ((\mathit{JSAST} \times D) \to \mathcal{O}((\mathit{JSAST} \times U))) \to (\mathit{JSAST} \to \mathit{JSAST})$$

The polymorphic types D and U represent information that is passed downward and upward, respectively, for a given pass. The individual parameters of `makePass` are described below:

1. Some default information to pass downward. This is used to seed the pass with downward information.
2. Some default information to pass upward. This is used in forwarding information up the AST when leaf nodes are encountered.
3. A function for combining two pieces of upward information into a single piece, used at internal AST nodes to coalesce upward information from child nodes. For example, if a pass is passing sets upward, then a possible function for combining sets is set union ($\cup$). This function is referred to as `combiner`.
4. A function that is called on the AST that results from the recursive application of translation process. This is the finishing function mentioned in the previous section, allowing for a handle on the global scope. The function takes both the processed AST and any information that was passed all the way up the AST, returning the final AST.
5. A partial function that performs the actual transformation work that is specific to the pass. It is defined only on AST nodes of interest to the pass, specifically returning **none** on AST nodes it is not defined on. If it is defined on a given AST node, it returns the transformed version of the node along with any information that needs to be forwarded up the AST. By convention, if a transformer is specified on a specific AST node instead of the more generic *JSAST*, then it is not defined over any other AST node. Conversely, if a transformer is specified on a specific AST node, then its result is always defined, and so the **some** constructor can be omitted.

The result of the `makePass` function is a function that will transform some input AST into a transformed version of said AST.

Another crucial helper function is `orElse`, which is used to chain two partial functions together. The semantics are that if the first partial function is not defined on the given input, then it tries the second partial function. If the second partial function is also not defined, then the whole result of `orElse` is also not defined. The definition for `orElse` is shown below:

$$\begin{aligned}
&\texttt{orElse} \in (A \to \mathcal{O}(B)) \times (A \to \mathcal{O}(B)) \to (A \to \mathcal{O}(B))\\
&\texttt{orElse}(f_1, f_2) =\\
&\quad a \Rightarrow\\
&\qquad \text{let } o_b = f_1(a)\\
&\qquad \begin{cases} o_b & \text{if } o_b \neq \textbf{none}\\ f_2(a) & \text{otherwise}\end{cases}
\end{aligned}$$

Related to `orElse` is the helper function `compose`, which simply chains together an arbitrary number of partial functions into a single partial function, utilizing `orElse` in the process. By convention, if multiple arguments are passed to `compose`, then these should be treated as a list in the same order as the arguments. The type signature and definition of `compose` is provided below:

$$\begin{aligned}
&\texttt{compose} \in \overrightarrow{(A \to \mathcal{O}(B))} \to (A \to \mathcal{O}(B))\\
&\texttt{compose}(\vec{f}) = \texttt{foldRight}(\texttt{orElse}, _ \Rightarrow \textbf{none}, \vec{f})
\end{aligned}$$

There also exists a special helper function, `transform`, which is used to recursively apply the transformation process. The type signature for `transform` is shown below:

$$\texttt{transform} \in A \times D \to A \times U$$

The polymorphic type A corresponds to some type of *JSAST*. For example, when applied to *JSStmt*, it returns another *JSStmt*. `transform` also takes some information to pass down the AST (D), and it returns some information to pass up the AST (U). The semantics of `transform` is that it first tries to transform the given AST node and downwards information with the partial function that is specific to a given pass. If the partial function is not defined for the AST node and downwards information, then it instead applies the default transformation described in the "Intuition" section.

Another helper function that is specific to individual passes is that of `combine`, which combines pieces of information which have been passed upward. Altogether, `combine` simply generalizes the user-defined `combiner` to an arbitrary number of upward inputs, using `defaultUpward` as the user-defined seed value that should be passed upwards. The type signature and definition for `combine` is detailed below:

$$\texttt{combine} \in \vec{U} \rightarrow U$$

$$\texttt{combine}(\vec{u}) = \texttt{foldLeft}(\texttt{combiner}, \texttt{defaultUpward}, \vec{u})$$

8.3 Pass Formalization

Formalization of individual passes follows. With the exception of the **Replace Empty With Undef** pass, which is provided as a gentle example of a pass formulation, only passes of significant complexity have been formalized. All passes are specified in terms of their arguments to `makePass`.

8.3.1 Replace Empty With Undef

This pass does not need to pass any information up or down, and so *Unit* is used as a filler for both U and D in `makePass`. Similarily, `combiner` also simply returns *Unit*. This pass does not need a handle on the global scope, so the finishing function merely returns the AST from the partial function like so:

$$\texttt{replaceEmptyWithUndefFinisher} \in \mathit{JSAST} \times \mathit{Unit} \rightarrow \mathit{JSAST}$$

$$\texttt{replaceEmptyWithUndefFinisher}(\mathit{jsast}, \mathit{Unit}) = \mathit{jsast}$$

The transformer itself simply returns `undef` whenever $\varnothing$ is encountered, shown below:

$$\texttt{replaceEmptyWithUndef} \in \mathit{JSAST} \times \mathit{Unit} \rightarrow \mathcal{O}(\mathit{JSAST} \times \mathit{Unit})$$

$$\texttt{replaceEmptyWithUndef}(\varnothing, \mathit{Unit}) = (\textsf{\textbf{undef}}, \mathit{Unit})$$

Overall the whole specification for this pass is:

$$\texttt{replaceEmptyWithUndefPass} \in \mathit{JSAST} \rightarrow \mathit{JSAST}$$

$$\texttt{replaceEmptyWithUndefPass} \ = \\ \quad \texttt{makePass}(\mathit{Unit}, \mathit{Unit}, (\mathit{Unit}, \mathit{Unit}) \Rightarrow \mathit{Unit}, \texttt{replaceEmptyWithUndefFinisher}, \texttt{replaceEmptyWithUndef})$$

8.3.2 Hoist Variable Declarations

This pass does not pass down any useful information, so *Unit* is used instead. Variables that need to be hoisted to their enclosing scope are passed upward in sets. For this reason, $\emptyset$ is used as the default upwards information, and set union ($\cup$) is used to combine upwards information.

This pass needs a handle on the global scope in order to set various properties of `window` to **undef**, depending on which global variables the program makes use of. The finishing function that handles this behavior is defined below:

$$\texttt{hoistVariableDeclarationsFinsiher} \in \mathit{JSAST} \times \overline{\mathit{JSVar}} \rightarrow \mathit{JSAST}$$

$$\texttt{hoistVariableDeclarationsFinisher}(\mathit{jsast}, \bar{x}) = \\ \quad \text{let } (\overline{y_{temp}}, \overline{x_{prog}}) = \texttt{partition}(\texttt{isTempVar}, \bar{x}) \\ \quad \text{let } \vec{s} = \texttt{listMap}(x \Rightarrow \texttt{window}[\texttt{varAsString}(x)] = \textsf{\textbf{undef}}, \texttt{asList}(\overline{x_{prog}})) \\ \quad \langle \overline{y_{temp}}, \texttt{asSeqJS}(\vec{s} + +(\mathit{jsast} :: \textsf{\textbf{nil}}))\rangle$$

Several partial functions are utilized to form a composite partial function, detailed below:

$\texttt{declHandler} \in \mathit{JSAST} \times \mathit{Unit} \to \mathcal{O}(\mathit{JSAST} \times \overline{\mathit{JSVar}})$

$\texttt{declHandler}((\texttt{var}\ \overrightarrow{x = o_e}), \mathit{Unit}) =$

$\quad \text{let } \bar{x} = \texttt{foldLeft}((\bar{x}', (x, \_)) \Rightarrow \bar{x}' \cup \{x\}, \emptyset, \overrightarrow{(x, o_e)})$

$\quad \texttt{doBinding} \in \mathit{JSVar} \times \mathit{JSExp} \to \mathit{JSStmt} \times \overline{\mathit{JSVar}}$

$\quad \text{let } \texttt{doBinding}\ x\ e =$

$\quad\quad \text{let } (e', \bar{x}) = \texttt{transform}(e, \mathit{Unit})$

$\quad\quad (x := e', \bar{x})$

$\quad \text{let } \overrightarrow{o_{\langle s, \bar{x}\rangle}} = \texttt{listMap}((x, o_e) \Rightarrow \texttt{optionMap}(e \Rightarrow \texttt{doBinding}(x, e), o_e), \overrightarrow{(x, o_e)})$

$\quad (\texttt{asSeqJS}(\texttt{listMapOption}(\texttt{fst}, \overrightarrow{o_{\langle s, \bar{x}\rangle}})),$

$\quad\ \texttt{flatten}(\texttt{listMapOption}(\texttt{snd}, \overrightarrow{o_{\langle s, \bar{x}\rangle}})) \cup \bar{x})$

$\texttt{transformDeclHandler} \in \mathit{JSAST} \times \mathit{Unit} \to \mathcal{O}(\mathit{JSAST} \times \overline{\mathit{JSVar}})$

$\texttt{transformDeclHandler}((\texttt{tempvar}\ \overrightarrow{y = e}\ e'), \mathit{Unit}) =$

$\quad \text{let } (\vec{e}, \vec{\bar{y}}) = \texttt{unzip}(\texttt{listMap}((\_, e) \Rightarrow \texttt{transform}(e, \mathit{Unit}), \overrightarrow{(y, e)}))$

$\quad \text{let } (e'', \bar{x}') = \texttt{transform}(e', \mathit{Unit})$

$\quad \text{let } \bar{x}'' = \texttt{flatten}(\vec{\bar{y}}) \cup \bar{x}'$

$\quad \text{let } \vec{y}''' = \texttt{listMap}(\texttt{fst}, \overrightarrow{(y, e)})$

$\quad (\texttt{foldRight}(((y, e_1), e_2) \Rightarrow ((y = e_1), e_2), e'', \texttt{zip}(\vec{y}''', \vec{e})),$

$\quad\ \bar{x}'' \cup \texttt{asSet}(\vec{y}'''))$

$\texttt{functionExpHandler} \in \mathit{JSAST} \times \mathit{Unit} \to \mathcal{O}(\mathit{JSAST} \times \overline{\mathit{JSVar}})$

$\texttt{functionExpHandler}((\texttt{fun}\ o_{str}\ \vec{x}\ s), \mathit{Unit}) =$

$\quad \text{let } (s_{body}, \overline{x_{body}}) = \texttt{transform}(s, \mathit{Unit})$

$\quad \texttt{makeFunction} \in \mathcal{O}(\mathit{JSVar}, \mathcal{O}(\mathit{JSExp})) \to \mathit{JSExp}$

$\quad \text{let } \texttt{makeFunction}(o_{\langle x, o_e\rangle}) =$

$\quad\quad \text{let } \overrightarrow{(x', o_e')} = \texttt{listMap}(x \Rightarrow (x, \textsf{\textbf{some undef}}), \texttt{asList}(\overline{x_{body}}))$

$$\text{let } \overrightarrow{(x'', o_e'')} = \begin{cases} (x, o_e) :: \overrightarrow{(x', o_e')} & \text{if } o_{\langle x, o_e\rangle} \neq \textsf{\textbf{none}} \\ \overrightarrow{(x', o_e')} & \text{otherwise} \end{cases}$$

$\quad\quad \texttt{fun}\ o_{str}\ \vec{x}\ (\texttt{var}\ \overrightarrow{(x'', o_e'')}; s_{body})$

$\quad \texttt{namedHandler} \in \mathit{String} \to \mathit{JSExp} \times \overline{\mathit{JSVar}}$

$\quad \text{let } \texttt{namedHandler}(\mathit{str}) =$

$\quad\quad \text{let } y = y \text{ is fresh}$

$\quad\quad (y = \texttt{makeFunction}(\textsf{\textbf{some}}\ (\texttt{jsVarToNotJSVar}(\mathit{str}), \textsf{\textbf{some}}\ y)), \{y\})$

$$\begin{cases} \texttt{namedHandler}(\mathit{str}) & \text{if } o_{str} = \textsf{\textbf{some}}\ \mathit{str} \\ (\texttt{makeFunction}(\textsf{\textbf{none}}), \emptyset) & \text{otherwise} \end{cases}$$

$$\texttt{functionDeclHandler} \in \mathit{JSAST} \times \mathit{Unit} \rightarrow \mathcal{O}(\mathit{JSAST} \times \overline{\mathit{JSVar}})$$

$$\begin{aligned}
&\texttt{functionDeclHandler}((\texttt{fun}\ \mathit{str}\ \vec{x}\ s), \mathit{Unit}) = \\
&\quad \text{let } (s_{body}, \overline{x_{body}}) = \texttt{transform}(s, \mathit{Unit}) \\
&\quad \text{let } \overrightarrow{(x, o_e)} = \texttt{listMap}(x \Rightarrow (x, \textsf{\textbf{some undef}}), \texttt{asList}(\overline{x_{body}})) \\
&\quad (\texttt{fun}\ \mathit{str}\ \vec{x}\ (\texttt{var}\ \overrightarrow{x = o_e}; s_{body}), \{\texttt{jsVarToNotJSVar}(\mathit{str})\})
\end{aligned}$$

The final partial function which handles the whole transformation is simply composed of all the previous partial functions, like so:

$$\begin{aligned}
&\texttt{hoistVariableDeclarations} \in \mathit{JSAST} \times \mathit{Unit} \rightarrow \mathcal{O}(\mathit{JSAST} \times \overline{\mathit{JSVar}}) \\
&\texttt{hoistVariableDeclarations} = \\
&\quad \texttt{compose}(\texttt{declHandler}, \texttt{transformDeclHandler}, \texttt{functionExpHandler}, \texttt{functionDeclHandler})
\end{aligned}$$

Overall, the whole specification for this pass is the following:

$$\begin{aligned}
&\texttt{hoistVariableDeclarationsPass} \in \mathit{JSAST} \rightarrow \mathit{JSAST} \\
&\texttt{hoistVariableDeclarationsPass} = \\
&\quad \texttt{makePass}(\mathit{Unit}, \emptyset, \\
&\qquad\qquad (\bar{x}_1, \bar{x}_2) \Rightarrow \bar{x}_1 \cup \bar{x}_2, \\
&\qquad\qquad \texttt{hoistVariableDeclarationsFinisher}, \\
&\qquad\qquad \texttt{hoistVariableDeclarations})
\end{aligned}$$

8.3.3 Make Global Variables `window` Properties

The information passed down by this pass is a stack of sets describing which variables are in scope. Lists are used instead of explicit stacks for simplicity, making the overall type of D the following: $\overrightarrow{\overline{\mathit{JSVar}}}$. Initially, only `window` is in scope (both the program-accessible and inaccessible version). In other words, the initial downwards information is $(\{\texttt{window}, \mathit{progWindow}\}) :: \textsf{\textbf{nil}}$. No information is passed upwards, so Unit is used for this purpose. Similarily, the function for combining upwards information simply unconditionally returns Unit. This pass does not need a handle on the global scope, so the finishing function simply returns the AST, like so:

$$\begin{aligned}
&\texttt{makeGlobalVariablesWindowPropertiesFinisher} \in \mathit{JSAST} \times \mathit{Unit} \rightarrow \mathit{JSAST} \\
&\texttt{makeGlobalVariablesWindowPropertiesFinisher}(\mathit{jsast}, \mathit{Unit}) = \mathit{jsast}
\end{aligned}$$

The `isInScope` function is a helper unique to this pass, used to determine whether or not a given variable is in scope. A special case for `isInScope` is the JavaScript `arguments` variable, which is automatically introduced in function contexts. A scope depth of 2 is significant, since the initial depth is 1 from the introduction of `window`, and hitting the toplevel variable declaration increases the depth to 2. Therefore, if the depth is greater than 2, it must mean that the translation is nested within a function. The entire `isInScope` function is detailed below:

$$\begin{aligned}
&\texttt{isInScope} \in \mathit{JSVar} \times \overrightarrow{\overline{\mathit{JSVar}}} \rightarrow \mathit{Boolean} \\
&\texttt{isInScope}(x, \vec{\bar{x}}) = \\
&\quad \texttt{checkStack} \in \overrightarrow{\overline{\mathit{JSVar}}} \rightarrow \mathit{Boolean} \\
&\quad \text{let } \texttt{checkStack}(\vec{\bar{x}}) = \begin{cases} \textsf{\textbf{false}} & \text{if } \texttt{isEmpty}(\vec{\bar{x}}) \\ \textsf{\textbf{true}} & \text{if } \neg\texttt{isEmpty}(\vec{\bar{x}}) \wedge x \in \texttt{head}(\vec{\bar{x}}) \\ \texttt{checkStack}(\texttt{tail}(\vec{\bar{x}})) & \text{otherwise} \end{cases} \\
&\quad \begin{cases} \textsf{\textbf{true}} & \text{if } \texttt{varAsString}(x) = \text{``}\mathit{arguments}\text{''} \wedge \texttt{length}(\vec{\bar{x}}) > 2 \\ \texttt{checkStack}(\vec{\bar{x}}) & \text{otherwise} \end{cases}
\end{aligned}$$

The pass is overall composed of several partial functions, detailed below:

$$\texttt{toplevelDeclHandler} \in \mathit{JSAST} \times \overrightarrow{\mathit{JSVar}} \to \mathcal{O}(\mathit{JSAST} \times \mathit{Unit})$$

$$\texttt{toplevelDeclHandler}((\langle \overline{x_{vars}}, s\rangle), \overrightarrow{x_{scope}}) =$$

$$\quad \text{let } (s', \_) = \texttt{transform}(s, \overline{x_{vars}} :: \overrightarrow{x_{scope}})$$

$$\quad (\langle \overline{x_{vars}}, s'\rangle, \mathit{Unit})$$

The previous pass guarantees that the first statement in the body of a function will always be a variable declaration. It also guarantees that variable declarations will only be encountered as the first statement in a function, and all functions will begin with a variable declaration.

$$\texttt{functionHandler} \in \mathit{JSAST} \times \overrightarrow{\mathit{JSVar}} \to \mathcal{O}(\mathit{JSAST} \times \mathit{Unit})$$

$$\texttt{functionHandler}(\mathit{jsast}, \vec{\vec{x}}) =$$

$$\quad \texttt{transformBody} \in \overrightarrow{\mathit{JSVar}} \times \mathit{JSStmt} \to \mathit{JSStmt}$$

$$\quad \text{let } \texttt{transformBody}(\overrightarrow{x_{func}}, s) =$$

$$\qquad \text{let } (\texttt{var } \overrightarrow{x = o_e}) :: \overrightarrow{s_{rest}} = s$$

$$\qquad \text{let } \overline{x_{decl}} = \texttt{asSet}(\texttt{listMap}(\texttt{fst}, \overrightarrow{(x, o_e)}))$$

$$\qquad \text{let } \vec{\vec{x}}' = (\overline{x_{decl}} \cup \texttt{asSet}(\overrightarrow{x_{func}})) :: \vec{\vec{x}}$$

$$\qquad \text{let } \vec{s} = \texttt{listMap}(s \Rightarrow \texttt{fst}(\texttt{transform}(s, \vec{\vec{x}}')), \overrightarrow{s_{rest}})$$

$$\qquad (\texttt{var } \overrightarrow{x = o_e}; \texttt{asSeqJS}(\vec{s}))$$

$$\begin{cases} \textsf{some } (\texttt{fun } \mathit{str}\ \vec{x}\ (\texttt{transformBody}(s)), \mathit{Unit}) & \text{if } \mathit{jsast} = \texttt{fun } \mathit{str}\ \vec{x}\ s \\ \textsf{some } (\texttt{fun } o_{str}\ \vec{x}\ (\texttt{transformBody}(s)), \mathit{Unit}) & \text{if } \mathit{jsast} = \texttt{fun } o_{str}\ \vec{x}\ s \\ \textsf{none} & \text{otherwise} \end{cases}$$

The previous pass guarantees that only simple assignments are encountered, as opposed to compound assignments (i.e. assignments annotated with a binary operator as in `x += y`).

$$\texttt{simpleAssignHandler} \in \mathit{JSAST} \times \overrightarrow{\mathit{JSVar}} \to \mathcal{O}(\mathit{JSAST} \times \mathit{Unit})$$

$$\texttt{simpleAssignHandler}((x = e), \vec{\vec{x}}) =$$

$$\quad \text{let } (e', \_) = \texttt{transform}(e, \vec{\vec{x}})$$

$$\quad \begin{cases} x = e' & \text{if } \texttt{isInScope}(x, \vec{\vec{x}}) \\ \texttt{window}[\texttt{varAsString}(x)] = e' & \text{otherwise} \end{cases}$$

$$\texttt{tryHandler} \in \mathit{JSAST} \times \overrightarrow{\mathit{JSVar}} \to \mathcal{O}(\mathit{JSAST} \times \mathit{Unit})$$

$$\texttt{tryHandler}(\texttt{try } s \texttt{ catch } o_{\langle x,s\rangle} \texttt{ fin } o_s) =$$

$$\quad \text{let } (s_{body}, \_) = \texttt{transform}(s, \vec{\vec{x}})$$

$$\quad \text{let } o'_{\langle x,s\rangle} = \texttt{optionMap}(((x'_{catch}, s'_{catch})) \Rightarrow (x'_{catch}, \texttt{fst}(\texttt{transform}(s'_{catch}, \{x'_{catch}\} :: \vec{\vec{x}}))), o_{\langle x,s\rangle})$$

$$\quad \text{let } o'_s = \texttt{optionMap}(s \Rightarrow \texttt{fst}(\texttt{transform}(s, \vec{\vec{x}})), o_s)$$

$$\quad ((\texttt{try } s_{body} \texttt{ catch } o'_{\langle x,s\rangle} \texttt{ fin } o'_s), \mathit{Unit})$$

$$\texttt{varHandler} \in \mathit{JSAST} \times \overrightarrow{\mathit{JSVar}} \to \mathcal{O}(\mathit{JSAST} \times \mathit{Unit})$$

$$\texttt{varHandler}(x, \vec{\vec{x}}) = \begin{cases} (x, \mathit{Unit}) & \text{if } \texttt{isInScope}(x, \vec{\vec{x}}) \\ (\texttt{window}[\texttt{varAsString}(x)], \mathit{Unit}) & \text{otherwise} \end{cases}$$

The final partial function is composed from the above partial functions like so:

$$\texttt{makeGlobalVariablesWindowProperties} \in \mathit{JSAST} \times \overrightarrow{\mathit{JSVar}} \rightarrow \mathcal{O}(\mathit{JSAST} \times \mathit{Unit})$$

$$\texttt{makeGlobalVariablesWindowProperties} =$$

$$\quad \texttt{compose}(\texttt{toplevelDeclHandler}, \texttt{varHandler}, \texttt{functionHandler}, \texttt{simpleAssignHandler}, \texttt{tryHandler})$$

Overall, the whole specification for this pass is the following:

$$\texttt{makeGlobalVariablesWindowPropertiesPass} \in \mathit{JSAST} \rightarrow \mathit{JSAST}$$

$$\texttt{makeGlobalVariablesWindowPropertiesPass} =$$

$$\quad \texttt{makePass}((\{\texttt{window}, \mathit{progWindow}\}) :: \textsf{nil},$$

$$\qquad \mathit{Unit}, (\mathit{Unit}, \mathit{Unit}) \Rightarrow \mathit{Unit},$$

$$\qquad \texttt{makeGlobalVariablesWindowPropertiesFinisher},$$

$$\qquad \texttt{makeGlobalVariablesWindowProperties})$$

8.3.4 Make All Assignments Simple

This pass does not need to pass any information either up or down, so *Unit* is used for both upwards and downwards values. The `combiner` function similarily returns *Unit*. Moreover, this pass does not need a handle on the global scope, and so the finishing function simply returns the AST it was passed.

A series of partial functions are composed to form this pass, which are detailed below:

$$\texttt{compoundAssignHandler} \in \mathit{JSAST} \times \mathit{Unit} \rightarrow \mathcal{O}(\mathit{JSAST} \times \mathit{Unit})$$

$$\texttt{compoundAssignHandler}((x \stackrel{\oplus}{=} e), \mathit{Unit}) =$$

$$\quad (x = x \oplus \texttt{fst}(\texttt{transform}(e, \mathit{Unit})), \mathit{Unit})$$

$$\texttt{compoundUpdateHandler} \in \mathit{JSAST} \times \mathit{Unit} \rightarrow \mathcal{O}(\mathit{JSAST} \times \mathit{Unit})$$

$$\texttt{compoundUpdateHandler}((e_1[e_2] \stackrel{\oplus}{=} e_3), \mathit{Unit}) =$$

$$\quad \text{let } y_1 = y \text{ is fresh}$$

$$\quad \text{let } y_2 = y \text{ is fresh}$$

$$\quad \text{let } e_{obj} = \texttt{toObj}(\texttt{fst}(\texttt{transform}(e_1, \mathit{Unit})))$$

$$\quad \text{let } e_{field} = \texttt{fst}(\texttt{transform}(e_2, \mathit{Unit}))$$

$$\quad \text{let } e_{rhs} = \texttt{fst}(\texttt{transform}(e_3, \mathit{Unit}))$$

$$\quad ((\texttt{tempvar}\ ((y_1, e_{obj}) :: (y_2, e_{field}) :: \textsf{nil})(y_1[y_2] = y_1[y_2]\ \oplus\ e_{rhs})), \mathit{Unit})$$

The pass as a whole is detailed below:

$$\texttt{makeAllAssignmentsSimplePass} \in \mathit{JSAST} \rightarrow \mathit{JSAST}$$

$$\texttt{makeAllAssignmentsSimplePass} =$$

$$\quad \texttt{makePass}(\mathit{Unit}, \mathit{Unit}, (\mathit{Unit}, \mathit{Unit}) \Rightarrow \mathit{Unit},$$

$$\qquad (\mathit{jsast}, \mathit{Unit}) \Rightarrow \mathit{jsast},$$

$$\qquad \texttt{orElse}(\texttt{compoundAssignHandler}, \texttt{compoundUpdateHandler}))$$

8.3.5 Hoist Functions

This pass does not need to deal with any information flowing downward, so *Unit* is used for downward information. Information is passed upward in the form of function declarations that need to be hoisted to the nearest enclosing scope (specifically $\overrightarrow{JSStmt}$). The `combiner` function used is list concatenation. The finishing function needs to put the lifted function definitions in the global scope, like so:

$\texttt{hoistFunctionsFinisher} \in \mathit{JSAST} \times \overrightarrow{\mathit{JSStmt}} \rightarrow \mathit{JSAST}$

$\texttt{hoistFunctionsFinisher}(\mathit{jsast}, \vec{s}) =$

$\quad \texttt{asSeqJS}(\texttt{reverse}(\texttt{asStmt}(\mathit{jsast}) :: \vec{s}))$

It is guaranteed by the previous pass that the AST passed to the finishing function will be a statement, so the result of `asStmt` is always defined.

`newBody` is a helper function that is unique to this pass. `newBody` is detailed below:

$\texttt{newBody} \in \mathit{JSStmt} \rightarrow \mathit{JSStmt}$

$\texttt{newBody}(s) =$

$\quad \text{let } (s', \vec{s}) = \texttt{transform}(s, \mathit{Unit})$

$\quad \texttt{asSeqJS}(\texttt{reverse}(s' :: \vec{s}))$

This pass is composed of a series of partial functions, detailed below:

$\texttt{functionDeclHandler} \in \mathit{JSAST} \times \mathit{Unit} \rightarrow \mathcal{O}(\mathit{JSAST} \times \overrightarrow{\mathit{JSStmt}})$

$\texttt{functionDeclHandler}((\texttt{fun}\ \mathit{str}\ \vec{x}\ s), \mathit{Unit}) =$

$\quad (\textbf{undef}, ((\texttt{fun}\ \mathit{str}\ \vec{x}\ \texttt{newBody}(s)) :: \textbf{nil}))$

$\texttt{functionExpHandler} \in \mathit{JSAST} \times \mathit{Unit} \rightarrow \mathcal{O}(\mathit{JSAST} \times \overrightarrow{\mathit{JSStmt}})$

$\texttt{functionExpHandler}((\texttt{fun}\ o_{str}\ \vec{x}\ s), \mathit{Unit}) =$

$\quad ((\texttt{fun}\ o_{str}\ \vec{x}\ \texttt{newBody}(s)), \textbf{nil})$

The pass as a whole is formalized as follows:

$\texttt{hoistFunctionsPass} \in \mathit{JSAST} \rightarrow \mathit{JSAST}$

$\texttt{hoistFunctionsPass} =$

$\quad \texttt{makePass}(\mathit{Unit}, \textbf{nil},$

$\qquad (\vec{s_1}, \vec{s_2}) \Rightarrow \vec{s_1} ++ \vec{s_2},$

$\qquad \texttt{hoistFunctionsFinisher},$

$\qquad \texttt{orElse}(\texttt{functionDeclHandler}, \texttt{functionExpHandler}))$

9 JavaScript to **notJS**

The following describes the translation function, $\mathcal{T}$, which recursively converts a JavaScript AST into an equivalent **notJS** AST. Certain *JSAST* nodes have no translation, as they were completely eliminated in the JavaScript → JavaScript passes. As a reminder, these eliminated nodes are the following:

- Empty statements ($\varnothing$). These are eliminated by the **Replace Empty With Undef** pass.

- Compound assignments and updates (i.e. annotated assignments like `x += y`. These are eliminated by the **Make All Assignments Simple** pass.
- `this` expressions. These are removed by the **Remove `this`** pass.
- Function declarations (though **not** function expressions). These are removed by the **Function Declaration to Expression** pass.

9.1 Loops

9.1.1 While loop

$$
\begin{aligned}
&\mathcal{T}[\![\texttt{while}\ e\ s]\!] = \\
&\quad \text{let } (s_{guard}, e_{guard}, \bar{y_1}) = \mathcal{T}[\![e]\!] \\
&\quad \text{let } (s_{body}, \_, \bar{y_2}) = \mathcal{T}[\![s]\!] \\
&\quad \text{let } (s', e', \bar{y_3}) = \texttt{whileHelper}(s_{guard}, e_{guard}, e_{guard}, s_{body}, s_{guard}) \\
&\quad (s', e', \bar{y_1} \cup \bar{y_2} \cup \bar{y_3})
\end{aligned}
$$

9.1.2 Do-while loop

$$
\begin{aligned}
&\mathcal{T}[\![\texttt{do}\ s\ \texttt{while}\ e]\!] = \\
&\quad \text{let } (s_{guard}, e_{guard}, \bar{y_1}) = \mathcal{T}[\![e]\!] \\
&\quad \text{let } (s_{body}, \_, \bar{y_2}) = \mathcal{T}[\![s]\!] \\
&\quad \text{let } (s', e', \bar{y_3}) = \texttt{whileHelper}(\texttt{noOpStatement}, \textbf{true}, e_{guard}, s_{body}, s_{guard}) \\
&\quad (s', e', \bar{y_1} \cup \bar{y_2} \cup \bar{y_3})
\end{aligned}
$$

9.1.3 For loop

$$
\begin{aligned}
&\mathcal{T}[\![\texttt{for}\ s_1\ e\ s_2\ s_3]\!] = \\
&\quad \text{let } (s_{init}, \_, \bar{y_1}) = \mathcal{T}[\![s_1]\!] \\
&\quad \text{let } (s_{guard}, e_{guard}, \bar{y_2}) = \mathcal{T}[\![e]\!] \\
&\quad \text{let } (s_{inc}, \_, \bar{y_3}) = \mathcal{T}[\![s_2]\!] \\
&\quad \text{let } (s_{body}, \_, \bar{y_4}) = \mathcal{T}[\![s_3]\!] \\
&\quad \text{let } (s', e', \bar{y_5}) = \texttt{whileHelper}((s_{init}; s_{guard}), e_{guard}, e_{guard}, s_{body}, (s_{inc}; s_{guard})) \\
&\quad (s', e', \bar{y_1} \cup \bar{y_2} \cup \bar{y_3} \cup \bar{y_4} \cup \bar{y_5})
\end{aligned}
$$

9.1.4 For-in loop

$\mathcal{T}[\![\texttt{for}\ \mathit{lhs}\ \texttt{in}\ e\ s]\!] =$

let $(s_{obj}, e_{obj}, \bar{y_1}) = \mathcal{T}[\![e]\!]$

let $(s'_{obj}, e'_{obj}, \bar{y_2}) = \texttt{toObj}(e_{obj})$

let $(s_{body}, e_{body}, \bar{y_3}) = \mathcal{T}[\![s]\!]$

$\texttt{makeRetval} \in \mathit{Variable} \times \mathit{Stmt} \rightarrow \mathit{TranslationRetval}$

let $\texttt{makeRetval}(x, s) =$

let $s_{inner} =$

$(s;$

$\ell_{continue}:$

$s_{body})$

$(s_{obj}; s'_{obj};$

$\ell_{break}:$

$\textbf{for}\ x\ e'_{obj}\ s_{inner},$

$\textbf{undef}, \bar{y_1} \cup \bar{y_2} \cup \bar{y_3})$

$\texttt{accessHelper} \in \mathit{JSStmt} \times \mathit{JSStmt} \rightarrow \mathit{TranslationRetval}$

let $\texttt{accessHelper}(e_1, e_2) =$

let $(s_{aobj}, e_{aobj}, \bar{y_4}) = \mathcal{T}[\![e_1]\!]$

let $(s_{field}, e_{field}, \bar{y_5}) = \mathcal{T}[\![e_2]\!]$

let $(s_{access}, e'_{aobj}, e'_{field}, \bar{y_6}) = \texttt{accessSetup}(e_{aobj}, e_{field})$

let $y = y$ is fresh

let $s'_{inner} =$

$(s_{aobj}; s_{field}; s_{access};$

$e'_{aobj}.e'_{field} := y)$

let $(s', e', \bar{y_7}) = \texttt{makeRetval}(y, s'_{inner})$

$(s', e', \bar{y_4} \cup \bar{y_5} \cup \bar{y_6} \cup \bar{y_7})$

$$\begin{cases} \texttt{makeRetval}(\texttt{jsVarToNotJSVar}(x), \texttt{noOpStatement}) & \text{if } \mathit{lhs} = x \\ \texttt{accessHelper}(e_1, e_2) & \text{if } \mathit{lhs} = e_1.e_2 \end{cases}$$

9.2 Binary Operations

Type signatures and descriptions of nested helper functions and constants specific to binary operations follow:

- $\texttt{withStatements} \in \mathit{TranslationRetval} \rightarrow \mathit{TranslationRetval}$
  Incorporates the statements and temporary variables resulting from the evaluation of the left and right expressions in the operation into the final result.

- $\texttt{withStatementsExp} \in \mathit{Exp} \rightarrow \mathit{TranslationRetval}$
  A special version of `withStatements` for when no additional statements or temporary variables are needed in the final result.

- $\texttt{lessThanCore} \in \mathit{BinaryOp} \times \mathit{BinaryOp} \times \mathit{Bool} \rightarrow \mathit{TranslationRetval}$
  Holds commonality between $<$, $\leq$, $>$, and $\geq$. These operations can all be described in terms of a comparison operator specialized for strings, a comparison operator specialized for integers, and whether or not the left and right sides should be swapped, respectively. Swapping is an easy way to switch between greater than or less than.

- $\texttt{logicalHelper} \in \mathit{Exp} \rightarrow \mathit{TranslationRetval}$
  Holds commonality between logical `or` and logical `and` (|| and &&, respectively). It is intended that if whatever

the lefthand side evaluates to equals whatever the given expression evaluates to, then the result of evaluating the righthand side should be returned.

- `arithemticBinop` $\in$ *TranslationRetval*
  Handles the bulk of the arithmetic binary operations, with the notable exception of addition.

- `asPrimitive` $\in Exp \rightarrow \overline{Stmt} \times Exp \times \overline{Variable}$
  Helper specific to addition. If the given expression evaluates to a primitive, it simply returns it. Otherwise, it will call `valueOf` on the value to try to get a primitive. If the result of `valueOf` is not a primitive, then it will call `toString` on that. If the result still is not a primitive, then a type error is thrown.

- `additionBinop` $\in$ *TranslationRetval*
  Performs the addition binary operation (+).

- `inBinop` $\in$ *TranslationRetval*
  Performs the `in` binary operation.

- `instanceOfBinop` $\in$ *TranslationRetval*
  Performs the `instanceOf` binary operation.

$\mathcal{T}[\![e_1 \oplus e_2]\!] =$

let $(s_1', e_1', \bar{y_1}) = \mathcal{T}[\![e_1]\!]$

let $(s_2', e_2', \bar{y_2}) = \mathcal{T}[\![e_1]\!]$

let $\texttt{withStatements}(s, e, \bar{y}) =$

$(s_1'; s_2'; s, e, \bar{y_1} \cup \bar{y_2} \cup \bar{y})$

let $\texttt{withStatementsExp}(e) =$

$(s_1'; s_2', e, \bar{y_1} \cup \bar{y_2})$

let $\texttt{lessThanCore}(\oplus_1, \oplus_2, b) =$

let $(e_{left}, e_{right}) = \begin{cases} (e_2, e_1) & \text{if } b \\ (e_1, e_2) & \text{otherwise} \end{cases}$

let $(s_{leftnum}, e_{leftnum}, \overline{y_{leftnum}}) = \texttt{toNumber}(e_{left})$

let $(s_{rightnum}, e_{rightnum}, \overline{y_{rightnum}}) = \texttt{toNumber}(e_{right})$

let $y = y$ is fresh

$\texttt{withStatements}($

$\texttt{if}(\texttt{typeof}(e_{left}) === \text{``}string'' \ \&\& \ \texttt{typeof}(e_{right}) === \text{``}string''\text{)}\{$

$y := e_{left} \ \oplus_1 \ e_{right}$

$\}$ $\texttt{else}$ $\{$

$s_{leftnum}; s_{rightnum};$

$y := e_{leftnum} \ \oplus_2 \ e_{rightnum}$

$\},$

$y, \overline{y_{leftnum}} \cup \overline{y_{rightnum}} \cup \{y\})$

let $\texttt{logicalHelper}(e) =$

let $y = y$ is fresh

$(s_1';$

$\texttt{if}(\textbf{tobool}(e_1') === e)\{$

$s_2';$

$y := e_2'$

$\}$ $\texttt{else}$ $\{$

$y := e_1'$

$\},$

$y, \bar{y_1} \cup \bar{y_2} \cup \{y\})$

let $\texttt{arithmeticBinop} \ =$

let $(s_{left}, e_{left}, \overline{y_{left}}) = \texttt{toNumber}(e_1')$

let $(s_{right}, e_{right}, \overline{y_{right}}) = \texttt{toNumber}(e_2')$

$\texttt{withStatements}($

$s_{left}; s_{right},$

$e_{left} \ \texttt{JSBopToBop}(\oplus) \ e_{right},$

$\overline{y_{left}} \cup \overline{y_{right}})$

let $\texttt{asPrimitive}(e) =$

$\quad$ let $(s_{value}, e_{value}, \overline{y_{value}}) = \texttt{valueOf}(e)$

$\quad$ let $(s_{string}, e_{string}, \overline{y_{string}}) =$

$\quad\quad \texttt{toSomething}(e_{value}, e \Rightarrow e, \texttt{callToString})$

$\quad$ let $(s_{final}, e_{final}, \overline{y_{final}}) =$

$\quad\quad \texttt{toSomething}(e_{string}, e \Rightarrow e,$

$\quad\quad\quad\quad (\_, \_) \Rightarrow (\texttt{throwTypeError}, \texttt{undef}, \emptyset))$

$\quad (s_{value}; s_{string}; s_{final},$

$\quad e_{final}, \overline{y_{value}} \cup \overline{y_{string}} \cup \overline{y_{final}})$

let $\texttt{additionBinop} =$

$\quad$ let $(\vec{s''_1}, e''_1, \bar{y'_1}) = \texttt{asPrimitive}(e'_1)$

$\quad$ let $(\vec{s''_2}, e''_2, \bar{y'_2}) = \texttt{asPrimitive}(e'_2)$

$\quad$ let $y = y$ is fresh

$\quad \texttt{withStatements}($

$\quad\quad \texttt{asSeq}(\texttt{interleave}(\vec{s''_1}, \vec{s''_2}));$

$\quad\quad \texttt{if}(\texttt{typeof}(e''_1) === \text{``}string''))\{$

$\quad\quad\quad y := e''_1 \mathbin{++} (\textbf{tostr } e''_2)$

$\quad\quad \}\ \texttt{else}\ \{$

$\quad\quad\quad \texttt{if}(\texttt{typeof}(e''_2) === \text{``}string''))\{$

$\quad\quad\quad\quad y := (\textbf{tostr } e''_1) \mathbin{++} e''_2$

$\quad\quad\quad \}\ \texttt{else}\ \{$

$\quad\quad\quad\quad y := (\textbf{tonum } e''_1) + (\textbf{tonum } e''_2)$

$\quad\quad\quad \}$

$\quad\quad \},$

$\quad\quad y, \bar{y'_1} \cup \bar{y'_2} \cup \{y\})$

let $\texttt{inBinop} =$

$\quad$ let $(s_{string}, e_{string}, \overline{y_{string}}) = \texttt{toString}(e'_1)$

$\quad \texttt{withStatements}($

$\quad\quad s_{string};$

$\quad\quad \texttt{if}(\textbf{isprim}(e'_2))\{$

$\quad\quad\quad \texttt{throwTypeError}$

$\quad\quad \}\ \texttt{else}\ \{$

$\quad\quad\quad \texttt{noOpStatement}$

$\quad\quad \},$

$\quad\quad e_{string}\ \texttt{in}\ e'_2, \overline{y_{string}})$

let $\texttt{instanceOfBinop} =$

$\quad \texttt{withStatements}($

$\quad\quad \texttt{if}(\neg(\texttt{typeof}(e'_2) === \text{``}function''))\{$

$\quad\quad\quad \texttt{throwTypeError}$

$\quad\quad \}\ \texttt{else}\ \{$

$\quad\quad\quad \texttt{noOpStatement}$

$\quad\quad \},$

$\quad\quad e'_1\ \texttt{instanceOf}\ (e'_2.\text{``}prototype''), \emptyset)$

$$\begin{cases}
\texttt{arithmeticBinop} & \text{if } \oplus \in \{-, \times, \div, \%, \ll, \gg, \ggg, \&, |, \veebar\} \\
\texttt{additionBinop} & \text{if } \oplus = + \\
\texttt{withStatementsExp}(e'_1 === e'_2) & \text{if } \oplus = '===' \\
\texttt{withStatementsExp}(\neg(e'_1 === e'_2)) & \text{if } \oplus = '!==' \\
\texttt{withStatementsExp}(e'_1 == e'_2) & \text{if } \oplus = '==' \\
\texttt{withStatementsExp}(\neg(e'_1 == e'_2)) & \text{if } \oplus = '!=' \\
\texttt{lessThanCore}(\prec, <, \texttt{false}) & \text{if } \oplus = < \\
\texttt{lessThanCore}(\preceq, \leq, \texttt{false}) & \text{if } \oplus = \leq \\
\texttt{lessThanCore}(\prec, <, \texttt{true}) & \text{if } \oplus = > \\
\texttt{lessThanCore}(\preceq, \leq, \texttt{true}) & \text{if } \oplus = \geq \\
\texttt{inBinop} & \text{if } \oplus = \texttt{in} \\
\texttt{instanceOfBinop} & \text{if } \oplus = \texttt{instanceOf} \\
\texttt{withStatementsExp}(e'_2) & \text{if } \oplus = , \\
\texttt{logicalHelper}(\textbf{true}) & \text{if } \oplus = \&\& \\
\texttt{logicalHelper}(\textbf{false}) & \text{if } \oplus = ||
\end{cases}$$

9.3 Switch

Based strictly on the grammar, it appears that the translation allows for `switch` statements with multiple `default` clauses. However, such `switch` statements are implicitly disallowed in a manner external to the grammar.

The JavaScript `switch` construct is by far the most complex single form that is handled by the translator. Because of its high complexity, a series of examples are provided below to illustrate how the translation process works. These examples are JavaScript $\rightarrow$ semi-JavaScript translations, but they should add some clarity.

Consider a basic `switch` statement with no `default` clause and some overlapping `case`s, like so:

```
switch (foo) {
  case 1:
    x = 1;
    break;
  case 2:
    x = 2;
  case 3:
    x += 3;
}
```

This is translated like so:

```
var tempExp = foo;
var tempFallthrough = false;
break_label: {
  if (tempExp == 1 || tempFallthrough == true) {
    x = 1;
    tempFallthrough = true;
    jump break_label;
  }
  if (tempExp == 2 || tempFallthrough == true) {
    x = 2;
    tempFallthrough = true;
  }
  if (tempExp == 3 || tempFallthrough == true) {
    x += 3;
    tempFallthrough = true;
    jump break_label;
```

```
  }
}
```

In the above code, `tempExp` is simply a temporary variable holding the result of evaluating `foo`. This is needed since in an arbitrary `switch` statement, the value of `foo` could be mutated within. Additionally, based on the grammar, `foo` could be any arbitrary JavaScript expression.

The variable `tempFallthrough` tracks whether or not the execution has matched on a `case` before. This is needed in order to determine whether or not we should fallthrough to another `case`, as when a program matches on a `case` with no `break` statement within.

The label `break_label` is the concrete realization of ℓ_{break} in this example. This is to provide a point to jump out of the whole switch statement whenever `break` is encountered, as per the usual semantics of `switch`. The jump that is inserted at the end of the last `case` is unnecessary in this example, though it will become necessary in subsequent translation examples.

When `default` is in the tail position, the translation is slightly different. Consider the following JavaScript `switch` statement:

```
switch (foo) {
  case 1:
    x = 1;
    break;
  case 2:
    x = 2;
  default:
    x = -1;
}
```

This is translated like so:

```
var tempExp = foo;
var tempFallthrough = false;
break_label: {
  if (tempExp == 1 || tempFallthrough == true) {
    x = 1;
    tempFallthrough = true;
    jump break_label;
  }
  if (tempExp == 2 || tempFallthrough == true) {
    x = 1;
    tempFallthrough = true;
    x = -1;
    jump break_label;
  }
  x = -1;
}
```

As shown, the body of the `default` clause is duplicated - it is put both at the end of all the conditionals, and at the end of the last conditional. This is necessary for proper fallthrough behavior. This is also why a `break` is always inserted at the end of the last `case` as well: even if execution fellthrough into the `default` as opposed to jumping to the `default`, then the `default` clause should still only be executed once.

There are simpler ways to perform the translation when the `default` clause is in the tail position. The problem is that the `default` clause does not necessarily exist in the tail position. For example, consider the following JavaScript code, which has a `default` in a non-tail position:

```
switch (foo) {
  case 1:
    x = 1;
  default:
    x = -1;
  case 2:
```

```
    x += 2;
}
```

This is translated as such:

```
var tempExp = foo;
var tempFallthrough = false;
break_label: {
  if (tempExp == 1 || tempFallthrough == true) {
    x = 1;
    x = -1;
    tempFallthrough = true;
  }
  if (tempExp == 2 || tempFallthrough == true) {
    x += 2;
    tempFallthrough = true;
    jump break_label;
  }
  x = -1;
  x += 2;
}
```

Intuitively, there are two major steps that are specific to translation where the `default` clause is not in the tail position:

1. Wherever the `default` is, append its body to the end previous `case`, as long as there is a previous `case`. This will allow for fallthrough behavior to `default` from the previous `case`. If the previous `case` ends with a `break` statement, then the code appended will simply become dead code.

2. At the end of all the `case`s, put the body of the `default` clause, along with the ordered bodies of all the `case`s that followed the `default` clause in the original JavaScript code. This way, if no `case` matches, then the body of the `default` clause will be executed, and if there was fallthrough behavior in the original code then it will fallthrough to the appropriate `case`s. Once again, this can result in dead code in the prescence of program-defined `break` statements.

While this logic may appear complicated, this is the easiest approach known to the authors, at least without having the capability to perform a forward jump (labeled statements in JavaScript only allow for backward jumps).

Type signatures and descriptions of helper functions specific to the translation of `switch` are summarized below:

- `makeDefaultInTail` $\in \overrightarrow{\mathit{JSSwitchSegment}} \rightarrow \overrightarrow{\mathit{JSCase}} \times \mathcal{O}(\mathit{JSStmt})$
  Given a series of `switch` segments, `makeDefaultInTail` will transform them so that the `default` clause, if present, will be in the tail position. This translation is done in a manner that preserves the original semantics.

- `process` $\in \overrightarrow{\mathit{JSSwitchSegment}} \times \overrightarrow{\mathit{JSCase}} \times \mathcal{O}(\mathit{JSStmt}) \rightarrow \overrightarrow{\mathit{JSCase}} \times \mathcal{O}(\mathit{JSStmt})$
  Tail-recursive helper for `makeDefaultInTail` that performs the bulk of the work.

- `defaultHandler` $\in \overrightarrow{\mathit{JSCase}} \times \mathcal{O}(\mathit{JSStmt})$
  Helper for `process` that handles the case when a `switchDefault` clause has been encountered.

- `defaultHasExistingCases` $\in \overrightarrow{\mathit{JSCase}}$
  Helper for `defaultHandler` that handles the case when some `case`s have already been processed.

- `nilHandler` $\in \overrightarrow{\mathit{JSCase}} \times \mathcal{O}(\mathit{JSStmt})$
  Helper for `process` that is called when `process` finishes and encounters **nil**.

- `nilHasExistingCases` $\in \overrightarrow{\mathit{JSCase}}$
  Helper for `nilHandler` that handles the case when some `case`s have already been processed. For most `switch` statements this should be the case; the only exception is that of a `switch` statement that has no `case`s or `default` clause.

- `handleCase` $\in \mathit{JSCase} \rightarrow \overrightarrow{\mathit{Stmt}} \times \overline{\mathit{Variable}}$
  Given a `case`, `handleCase` returns a list of statements holding the `case`'s entire translation, along with a set of temporary variables used during translation.

- $\texttt{addCase} \in \mathit{JSCase} \times (\overrightarrow{\mathit{Stmt}} \times \overline{\mathit{Variable}}) \to \overrightarrow{\mathit{Stmt}} \times \overline{\mathit{Variable}}$
  Used as part of a `foldRight` operation during the translation of `switch`. Given some `case` and some preexisting result of building up other `case`s, it will add the result of the given `case` to the preexisting `case` results.

$$
\begin{array}{l}
\texttt{makeDefaultInTail}(\vec{w}) = \\
\quad \text{let } \texttt{process}(\vec{w}, \overrightarrow{case}, o_s) = \\
\quad\quad \text{let } \texttt{defaultHandler} = \\
\quad\quad\quad \text{let } (\texttt{switchDefault}\ s_{body}) = \texttt{head}(\vec{w}) \\
\quad\quad\quad \text{let } \texttt{defaultHasExistingCases} = \\
\quad\quad\quad\quad \text{let } (\texttt{switchCase}\ e\ s) = \texttt{head}(\overrightarrow{case}) \\
\quad\quad\quad\quad (\texttt{switchCase}\ e\ (s; s_{body})) :: \texttt{tail}(\overrightarrow{case}) \\
\quad\quad\quad \text{let } \overrightarrow{case}' = \begin{cases} \texttt{defaultHasExistingCases} & \text{if } \neg\texttt{isEmpty}(\overrightarrow{case}) \\ \overrightarrow{case} & \text{otherwise} \end{cases} \\
\quad\quad\quad \texttt{process}(\texttt{tail}(\vec{w}), \overrightarrow{case}', \textbf{some}\ (s_{body}; \texttt{asSeqJS}(\texttt{listMap}(\texttt{getBody}, \texttt{tail}(\vec{w}))))) \\
\quad\quad \text{let } \texttt{nilHandler} = \\
\quad\quad\quad \text{let } \texttt{nilHasExistingCases} = \\
\quad\quad\quad\quad \text{let } (\texttt{switchCase}\ e\ s) = \texttt{head}(\overrightarrow{case}) \\
\quad\quad\quad\quad (\texttt{switchCase}\ e\ (s; \texttt{break})) :: \texttt{tail}(\overrightarrow{case}) \\
\quad\quad\quad \text{let } \overrightarrow{case}' = \begin{cases} \texttt{nilHasExistingCases} & \text{if } \neg\texttt{isEmpty}(\overrightarrow{case}) \\ \overrightarrow{case} & \text{otherwise} \end{cases} \\
\quad\quad\quad (\texttt{reverse}(\overrightarrow{case}'), o_s) \\
\quad\quad \begin{cases} \texttt{nilHandler} & \text{if } \texttt{isEmpty}(\vec{w}) \\ \texttt{defaultHandler} & \text{if } \texttt{head}(\vec{w}) = \texttt{switchDefault}\ s \\ \texttt{process}(\texttt{tail}(\vec{w}), \texttt{head}(\vec{w}) :: \overrightarrow{case}, o_s) & \text{otherwise} \end{cases} \\
\quad \texttt{process}(\vec{w}, \textbf{nil}, \textbf{none})
\end{array}
$$

$\mathcal{T}[\![\texttt{switch}\ e\ \vec{w}]\!] =$

$\quad$ let $(s_1', e_1', \bar{y_1}') = \mathcal{T}[\![e]\!]$

$\quad$ let $(\overrightarrow{case}, o_s) = \texttt{makeDefaultInTail}(\vec{w})$

$\quad$ let $(s_2', e_2', \bar{y_2}') = \texttt{getOrElse}(\texttt{noOpStatementExp}(\textbf{undef}), \texttt{optionMap}(\mathcal{T}, o_s))$

$\quad$ let $y_{exp} = y$ is fresh

$\quad$ let $y_{fallthrough} = y$ is fresh

$\quad$ let $\texttt{handleCase}(\texttt{switchCase}\ e\ s) =$

$\quad\quad$ let $(s_3', e_3', \bar{y_3'}) = \mathcal{T}[\![e]\!]$

$\quad\quad$ let $(s_4', e_4', \bar{y_4'}) = \mathcal{T}[\![s]\!]$

$\quad\quad$ let $\texttt{ifPortion} =$

$\quad\quad\quad$ $\texttt{if}(e_3' === y_{exp}\ ||\ y_{fallthrough})\{$

$\quad\quad\quad\quad$ s_4';

$\quad\quad\quad\quad$ $y_{fallthrough} := \textbf{true}$

$\quad\quad\quad$ $\}\ \texttt{else}\ \{$

$\quad\quad\quad\quad$ $\texttt{noOpStatement}$

$\quad\quad\quad$ $\}$

$\quad\quad$ $(s_3' :: \texttt{ifPortion}, \bar{y_3'} \cup \bar{y_4'})$

$\quad$ let $\texttt{addCase}(case, (\vec{s}, \bar{y})) =$

$\quad\quad$ let $(\vec{s}', \bar{y}') = \texttt{handleCase}(case)$

$\quad\quad$ $(\vec{s}' + +\vec{s}, \bar{y}' \cup \bar{y})$

$\quad$ let $(\vec{s}'', \bar{y_4}) = \texttt{foldRight}(\texttt{addCase}, (\textbf{nil}, \emptyset), \overrightarrow{case})$

$\quad$ $(\ell_{break} :\ \{$

$\quad\quad$ s_1';

$\quad\quad$ $y_{exp} := e_1'$;

$\quad\quad$ $y_{fallthrough} := \textbf{false}$;

$\quad\quad$ $\texttt{asSeq}(\vec{s}'')$;

$\quad\quad$ s_2'

$\quad$ $\},$

$\quad$ $\textbf{undef}, \bar{y_1}' \cup \bar{y_2}' \cup \bar{y_4} \cup \{y_{cond}, y_{fallthrough}\})$

9.4 Functions

9.4.1 Function Expressions

The JS $\rightarrow$ JS passes guarantee that only function expressions will be encountered; no function declarations will exist in the source program. The JS $\rightarrow$ JS passes also guarantee that all functions begin with variable declarations, and the expressions within said declarations will either be undefined or a variable.

$\mathcal{T}[\![\texttt{fun}\ o_{str}\ \vec{x}\ s]\!] =$

$\quad$ let $((\texttt{var}\ \overrightarrow{x = o_e}) :: \overrightarrow{s_{rest}}) = s$

$\quad$ let $(\overrightarrow{s_{body}}, \_, \vec{\overline{y_1}}) = \texttt{unzip3}(\texttt{listMap}(\mathcal{T}, \overrightarrow{s_{rest}}))$

$\quad$ let $x_{args} = \texttt{stringAsVar}(``arguments'')$

$\quad$ let $i = \texttt{length}(\vec{x})$

$\quad$ $\texttt{argsBinding} \in \mathit{JSVar} \times \mathbb{Z} \rightarrow \mathit{Variable} \times \mathit{Exp}$

$\quad$ let $\texttt{argsBinding}(x, i') =$

$\quad\quad (\texttt{jsVarToNotJSVar}(x), x_{args}.\texttt{intAsString}(i'))$

$\quad$ let $\overrightarrow{(x', e')} = \texttt{listMap}(\texttt{argsBinding}, \texttt{zip}(\vec{x}, \texttt{range}(0, i-1)))$

$\quad$ $\texttt{expressionBinding} \in \mathcal{O}(\mathit{JSExp}) \rightarrow \mathit{Exp}$

$\quad$ let $\texttt{expressionBinding}(o_e) =$

$$\begin{cases} \textsf{undef} & \text{if } o_e = \textsf{some undef} \\ \texttt{jsVarToNotJSVar}(x) & \text{if } o_e = \textsf{some } x \end{cases}$$

$\quad$ let $\overrightarrow{(x'', e'')} = \texttt{listMap}((x, o_e) \Rightarrow (\texttt{jsVarToNotJSVar}(x), \texttt{expressionBinding}(o_e)), \overrightarrow{(x, o_e)})$

$\quad$ let $(s_{args}, e_{args}, \overline{y_{args}}) = \texttt{makeArguments}(\textsf{nil})$

$\quad$ let $\overrightarrow{(y_2, e''')} = \texttt{listMap}(y \Rightarrow (y, \textsf{undef}), \texttt{asList}(\texttt{flattenVars}(\vec{\overline{y_1}})))$

$\quad$ let $s_{decl} = \texttt{var}\ (\overrightarrow{(x', e')} + +\overrightarrow{(x'', e'')} + +\overrightarrow{(y_2, e''')})\ (l_{return} : \texttt{asSeq}(\overrightarrow{s_{body}}))$

$\quad$ let $y_{proto} = y$ is fresh

$\quad$ let $y_{retval} = y$ is fresh

$\quad$ $(y_{retval} := \textsf{newfun}\ ((\texttt{self}, x_{args}) \Rightarrow s_{decl})\ \texttt{asNum}(i);$

$\quad$ $s_{args};$

$\quad$ $y_{proto} := \texttt{new}\ x_{object}(e_{args});$

$\quad$ $y_{retval}.``prototype'' := y_{proto},$

$\quad$ $y_{retval}, \overline{y_{args}} \cup \{y_{proto}, y_{retval}\})$

9.4.2 Method Calls

$\mathcal{T}[\![e_1.e_2(\vec{e_3})]\!] =$

$\quad$ let $(s_{obj}, e_{obj}, \overline{y_{obj}}) = \mathcal{T}[\![e_1]\!]$

$\quad$ let $(s_{field}, e_{field}, \overline{y_{field}}) = \mathcal{T}[\![e_2]\!]$

$\quad$ let $(\overrightarrow{s_{args}}, \overrightarrow{e_{args}}, \overrightarrow{\overline{y_{args}}}) = \texttt{unzip3}(\texttt{listMap}(\mathcal{T}, \vec{e_3}))$

$\quad$ let $(s_{access}, e'_{obj}, e'_{field}, \overline{y_{access}}) = \texttt{accessSetup}(e_{obj}, e_{field})$

$\quad$ let $y = y$ is fresh

$\quad$ let $(s_{call}, \_, \overline{y_{call}}) = \texttt{call}(y, e'_{obj}.e'_{field}, e'_{obj}, \overrightarrow{e_{args}})$

$\quad$ $(s_{obj}; s_{field}; \texttt{asSeq}(\overrightarrow{s_{args}}); s_{access}; s_{call},$

$\quad$ $y,$

$\quad$ $\overline{y_{obj}} \cup \overline{y_{field}} \cup \texttt{flattenVars}(\overrightarrow{\overline{y_{args}}}) \cup \overline{y_{call}} \cup \overline{y_{access}} \cup \{y\})$

9.4.3 Function Calls

A precondition for function calls is that $e \neq e_1.e_2$, in order to avoid introducing nondeterminism with method calls.

$\mathcal{T}[\![e(\vec{e})]\!] =$

let $(s_{func}, e_{func}, \overline{y_{func}}) = \mathcal{T}[\![e]\!]$

let $(\overrightarrow{s_{param}}, \overrightarrow{e_{param}}, \overrightarrow{\overline{y_{param}}}) = \texttt{unzip3}(\texttt{listMap}(\mathcal{T}, \vec{e}))$

let $y = y$ is fresh

let $(s_{call}, \_, \overline{y_{call}}) = \texttt{call}(y, e_{func}, \texttt{window}, \overrightarrow{e_{param}})$

$(s_{func}; \texttt{asSeq}(\overrightarrow{s_{param}}); s_{call},$

$y,$

$\overline{y_{func}} \cup \texttt{flattenVars}(\overrightarrow{\overline{y_{param}}}) \cup \overline{y_{call}} \cup \{y\})$

9.5 Label-Related Routines

9.5.1 Labeled Statements

$\mathcal{T}[\![\vec{\ell}\, s]\!] =$

let $(s', \_, \bar{y}) = \mathcal{T}[\![s]\!]$

let $s''' = \texttt{foldRight}((\ell, s'') \Rightarrow \ell :\ s'', s', \vec{\ell})$

$(s''', \textsf{undef}, \bar{y})$

9.5.2 Break

$\mathcal{T}[\![\texttt{break}\ o_\ell]\!] =$

$(\textsf{jump}\ \texttt{optionLabel}(o_\ell, \ell_{break})\ \textsf{undef},$

$\textsf{undef}, \emptyset)$

9.5.3 Continue

$\mathcal{T}[\![\texttt{continue}\ o_\ell]\!] =$

$(\textsf{jump}\ \texttt{optionLabel}(o_\ell, \ell_{continue})\ \textsf{undef},$

$\textsf{undef}, \emptyset)$

9.5.4 Return

$\mathcal{T}[\![\texttt{return}\ o_e]\!] =$

let $(s, e', \bar{y}) = \mathcal{T}[\![\texttt{getOrElse}(\textsf{undef}, o_e)]\!]$

$(s; \textsf{jump}\ \ell_{return}\ e',$

$\textsf{undef}, \bar{y})$

9.6 Toplevel Declaration

It is guaranteed by the JavaScript → JavaScript passes that the AST will start with a toplevel declaration by the time it reaches $\mathcal{T}$.

$\mathcal{T}[\![\langle \bar{x}, s\rangle]\!] =$

$\quad$ let $\overrightarrow{(x, e)} = \texttt{listMap}(x \Rightarrow (\texttt{jsVarToNotJSVar}(x), \textsf{undef}), \texttt{asList}(\bar{x}))$

$\quad$ let $(s', \_, \bar{y}) = \mathcal{T}[\![s]\!]$

$\quad$ let $\overrightarrow{(y', e')} = \texttt{listMap}(x \Rightarrow (x, \textsf{undef}), \texttt{asList}(\bar{y}))$

$\quad$ $(\textsf{decl}\ \overrightarrow{(x, e)} ++ \overrightarrow{(y', e')}\ \textsf{in}\ s', \textsf{undef}, \emptyset)$

9.7 Object Update

$\mathcal{T}[\![e_1[e_2] = e_3]\!] =$

$\quad$ let $(s_{obj}, e_{obj}, \overline{y_{obj}}) = \mathcal{T}[\![e_1]\!]$

$\quad$ let $(s_{field}, e_{field}, \overline{y_{field}}) = \mathcal{T}[\![e_2]\!]$

$\quad$ let $(s_{rhs}, e_{rhs}, \overline{y_{rhs}}) = \mathcal{T}[\![e_3]\!]$

$\quad$ let $(s_{access}, e'_{obj}, e'_{field}, \overline{y_{access}}) = \texttt{accessSetup}(e_{obj}, e_{field})$

$\quad$ $(s_{rhs}; s_{obj}; s_{field}; s_{access}; e_{obj}.e_{field} = e_{rhs},$

$\quad$ $e_{rhs},$

$\quad$ $\overline{y_{obj}} \cup \overline{y_{field}} \cup \overline{y_{rhs}} \cup \overline{y_{access}})$

9.8 Values and Variables

All the constants in JavaScript are being conflated with all the constants in **notJS** here, specifically *JSNum*(n), *JSBool* (b), *JSStr* (str), **undef**, and **null**. While the syntax shows that these are distinct, the separation is largely needless, and so it is considered acceptable to have this sort of conflation.

$\mathcal{T}[\![n]\!] = \texttt{noOpStatementExp}(n)$

$\mathcal{T}[\![b]\!] = \texttt{noOpStatementExp}(b)$

$\mathcal{T}[\![s]\!] = \texttt{noOpStatementExp}(s)$

$\mathcal{T}[\![\textsf{undef}]\!] = \texttt{noOpStatementExp}(\textsf{undef})$

$\mathcal{T}[\![\textsf{null}]\!] = \texttt{noOpStatementExp}(\textsf{null})$

$\mathcal{T}[\![x]\!] = \texttt{noOpStatementExp}(\texttt{jsVarToNotJSVar}(x))$

9.9 Variable Assignment

$\mathcal{T}[\![x = e]\!] =$

$\quad$ let $(s, e', \bar{y}) = \mathcal{T}[\![e]\!]$

$\quad$ $(s; \texttt{jsVarToNotJSVar}(x) := e',$

$\quad$ $e', \bar{y})$

9.10 Regular Expression Literals

$\mathcal{T}[\![\texttt{regexp}\ str\ o_{str}]\!] =$

$$\text{let } \overrightarrow{str} = \begin{cases} str' :: \textsf{nil} & \text{if } o_{str} = \textsf{some } str' \\ \textsf{nil} & \text{otherwise} \end{cases}$$

let $(s, e, \bar{y}) = \texttt{makeArguments}(str :: \overrightarrow{str})$

let $y = y$ is fresh

$(s; y := \textsf{new } x_{regex}(e), y, \bar{y} \cup \{y\})$

9.11 Ternary Operator

$\mathcal{T}[\![e_1\,?\,e_2 : e_3]\!] =$

let $(s_{guard}, e_{guard}, \overline{y_{guard}}) = \mathcal{T}[\![e_1]\!]$

let $(s_{true}, e_{true}, \overline{y_{true}}) = \mathcal{T}[\![e_2]\!]$

let $(s_{false}, e_{false}, \overline{y_{false}}) = \mathcal{T}[\![e_3]\!]$

let $y = y$ is fresh

$(s_{guard};$

if($\textsf{tobool}(e_{guard})$){

$s_{true};\ y := e_{true}$

} else {

$s_{false};\ y := e_{false}$

},

$y, \overline{y_{guard}} \cup \overline{y_{true}} \cup \overline{y_{false}} \cup \{y\})$

9.12 Object Access

$\mathcal{T}[\![e_1.e_2]\!] =$

let $(s_{obj}, e_{obj}, \overline{y_{obj}}) = \mathcal{T}[\![e_1]\!]$

let $(s_{field}, e_{field}, \overline{y_{field}}) = \mathcal{T}[\![e_2]\!]$

let $(s_{access}, e'_{obj}, e'_{field}, \overline{y_{access}}) = \texttt{accessSetup}(e_{obj}, e_{field})$

$(s_{obj}; s_{field}; s_{access},$

$e'_{obj}.e'_{field},$

$\overline{y_{obj}} \cup \overline{y_{field}} \cup \overline{y_{access}})$

9.13 New

$\mathcal{T}[\![\texttt{new}\ e(\vec{e})]\!] =$

$\quad$ let $(s_{obj}, e_{obj}, \overline{y_{obj}}) = \mathcal{T}[\![e]\!]$

$\quad$ let $(\overrightarrow{s_{param}}, \overrightarrow{e_{param}}, \overrightarrow{\overline{y_{param}}}) = \texttt{unzip3}(\texttt{listMap}(\mathcal{T}, \vec{e}))$

$\quad$ let $(s_{args}, e_{args}, \overline{y_{args}}) = \texttt{makeArguments}(\overrightarrow{e_{param}})$

$\quad$ let $y = y$ is fresh

$\quad$ $(s_{obj}; \texttt{asSeq}(\overrightarrow{s_{param}});$

$\quad$ if($\textsf{\textbf{isprim}}(e_{obj})$) {

$\qquad$ `throwTypeError`

$\quad$ } else {

$\qquad$ $s_{args}; y := \texttt{new}\ e_{obj}(e_{args})$

$\quad$ },

$\quad$ $y, \overline{y_{obj}} \cup \texttt{flattenVars}(\overrightarrow{\overline{y_{param}}}) \cup \overline{y_{args}} \cup \{y\})$

9.14 Unary Operators

$\mathcal{T}[\![\odot\ e]\!] =$

$\quad$ let $(s, e', \bar{y}) = \mathcal{T}[\![e]\!]$

$\quad$ $\texttt{withStatement} \in \mathit{TranslationRetval} \rightarrow \mathit{TranslationRetval}$

$\quad$ let $\texttt{withStatement}(s', e'', \bar{y'}) =$

$\qquad$ $(s; s', e'', \bar{y} \cup \bar{y'})$

$\quad$ $\texttt{withStatementTup} \in (\mathit{TranslationRetval}) \rightarrow \mathit{TranslationRetval}$

$\quad$ let $\texttt{withStatementTup}((s', e'', \bar{y'})) =$

$\qquad$ $\texttt{withStatement}(s', e'', \bar{y'})$

$\quad$ $\texttt{withStatementExp} \in \mathit{Exp} \rightarrow \mathit{TranslationRetval}$

$\quad$ let $\texttt{withStatementExp}(e'') =$

$\qquad$ $(s, e'', \bar{y})$

$\quad$ let $\texttt{negationHelper}\ =$

$\qquad$ let $(s_{num}, e_{num}, \overline{y_{num}}) = \texttt{toNumber}(e')$

$\qquad$ $\texttt{withStatement}(s_{num}, \odot\ e_{num}, \overline{y_{num}})$

$$\begin{cases} \texttt{withStatementExp}(\textsf{\textbf{undef}}) & \text{if } \odot = \texttt{void} \\ \texttt{withStatementExp}(\texttt{typeOf}(e')) & \text{if } \odot = \texttt{typeof} \\ \texttt{withStatementTup}(\texttt{toNumber}(e')) & \text{if } \odot = + \\ \texttt{negationHelper} & \text{if } \odot \in \{-, \sim\} \\ \texttt{withStatementExp}(\neg\textsf{\textbf{tobool}}(e')) & \text{if } \odot = \neg \\ \texttt{withStatementTup}(\textsf{\textbf{toobj}}(e')) & \text{if } \odot = \texttt{toObj} \end{cases}$$

9.15 Object Literals

$\mathcal{T}[\![\{\overrightarrow{str: e}\}]\!] =$

$\quad$ $\texttt{processField} \in \mathit{String} \times \mathit{Exp} \rightarrow \mathit{String} \times \mathit{Stmt} \times \mathit{Exp} \times \overrightarrow{\mathit{Variable}}$

$\quad$ let $\texttt{processField}(str, e) =$

$\quad\quad$ let $(s_{field}, e_{field}, \overline{y_{field}}) = \mathcal{T}[\![e]\!]$

$\quad\quad$ $(str, s_{field}, e_{field}, \overline{y_{field}})$

$\quad$ let $(\overrightarrow{str_{field}}, \overrightarrow{s_{field}}, \overrightarrow{e_{field}}, \overrightarrow{\overline{y_{field}}}) = \texttt{unzip4}(\texttt{listMap}(\texttt{processField}, \overrightarrow{(str, e)}))$

$\quad$ let $(s_{args}, e_{args}, \overline{y_{args}}) = \texttt{makeArguments}(\textbf{nil})$

$\quad$ let $y = y$ is fresh

$\quad$ $\texttt{addBinding} \in \mathit{String} \times \mathit{Exp} \rightarrow \mathit{Stmt}$

$\quad$ let $\texttt{addBinding}(str, e) =$

$\quad\quad$ $y.str := e$

$\quad$ $(\texttt{asSeq}(\overrightarrow{s_{field}});$

$\quad$ $s_{args};$

$\quad$ $y := \texttt{new}\ x_{object}(e_{args});$

$\quad$ $\texttt{asSeq}(\texttt{listMap}(\texttt{addBinding}, \texttt{zip}(\overrightarrow{str_{field}}, \overrightarrow{e_{field}}))),$

$\quad$ $y, \texttt{flattenVars}(\overrightarrow{\overline{y_{field}}}) \cup \overline{y_{args}} \cup \{y\})$

9.16 Array Literals

$\mathcal{T}[\![[\vec{e}]]\!] =$

$\quad$ let $(\vec{s}, \vec{e'}, \vec{\bar{y}}) = \texttt{unzip3}(\texttt{listMap}(\mathcal{T}, \vec{e}))$

$\quad$ let $(s_{args}, e_{args}, \bar{y}') = \texttt{makeArguments}(\textbf{nil})$

$\quad$ let $y = y$ is fresh

$\quad$ $\texttt{updatePosition} \in \mathit{Exp} \times \mathbb{Z} \rightarrow \mathit{Stmt}$

$\quad$ let $\texttt{updatePosition}(e, i) =$

$\quad\quad$ $y.\texttt{intAsString}(i) := e$

$\quad$ let $i = \texttt{length}(\vec{e'})$

$\quad$ $(\texttt{asSeq}(\vec{s});$

$\quad$ $s_{args};$

$\quad$ $y := \texttt{new}\ x_{array}(e_{args});$

$\quad$ $\texttt{asSeq}(\texttt{listMap}(\texttt{updatePosition}, \texttt{zip}(\vec{e'}, \texttt{range}(0, i-1))));$

$\quad$ $y.\text{“}length'' := \texttt{asNum}(i),$

$\quad$ $y, \texttt{flattenVars}(\vec{\bar{y}}) \cup \bar{y}' \cup \{y\})$

9.17 Prefix/Postfix Increment/Decrement

Our syntax only permits one to use prefix/postfix increment/decrement on a *JSLHS*, so only x and $e_1.e_2$ need to be considered as the expresion being incremented/decremented.

9.17.1 Prefix Increment Variable

$\mathcal{T}[\![++\,x]\!] = \texttt{prefixVarHelper}(x, 1)$

9.17.2 Prefix Increment Access

$\mathcal{T}[\![++\,e_1.e_2]\!] = \texttt{prefixAccessHelper}(e_1, e_2, 1)$

9.17.3 Postfix Increment Variable

$\mathcal{T}[\![x\,++]\!] = \texttt{postfixVarHelper}(x, 1)$

9.17.4 Postfix Increment Access

$\mathcal{T}[\![e_1.e_2\,++]\!] = \texttt{postfixAccessHelper}(e_1, e_2, 1)$

9.17.5 Prefix Decrement Variable

$\mathcal{T}[\![--\,x]\!] = \texttt{prefixVarHelper}(x, -1)$

9.17.6 Prefix Decrement Access

$\mathcal{T}[\![--\,e_1.e_2]\!] = \texttt{prefixAccessHelper}(e_1, e_2, -1)$

9.17.7 Postfix Decrement Variable

$\mathcal{T}[\![x\,--]\!] = \texttt{postfixVarHelper}(x, -1)$

9.17.8 Postfix Decrement Access

$\mathcal{T}[\![e_1.e_2\,--]\!] = \texttt{postfixAccessHelper}(e_1, e_2, -1)$

9.18 Sequence of Statements

$$
\begin{aligned}
&\mathcal{T}[\![\vec{s}]\!] = \\
&\quad \text{let } \texttt{hasStatements} \ = \\
&\quad\quad \text{let } (\vec{s'}, \vec{e}, \vec{\vec{y}}) = \texttt{unzip3}(\texttt{listMap}(\mathcal{T}, \vec{s})) \\
&\quad\quad (\texttt{asSeq}(\vec{s'}), \texttt{last}(\vec{e}), \texttt{flattenVars}(\vec{\vec{y}})) \\
&\quad \begin{cases} \texttt{noOpStatementExp}(\textbf{undef}) & \text{if } \texttt{isEmpty}(\vec{s}) \\ \texttt{hasStatements} & \text{otherwise} \end{cases}
\end{aligned}
$$

9.19 Conditionals

$\mathcal{T}[\![\texttt{if}\ e\ s\ o_s]\!]$

$\quad$ let $(s_{guard}, e_{guard}, \overline{y_{guard}}) = \mathcal{T}[\![e]\!]$

$\quad$ let $(s_1, \_, \bar{y_1}) = \mathcal{T}[\![s]\!]$

$\quad$ let $(s_2, \_, \bar{y_2}) = \mathcal{T}[\![\texttt{getOrElse}(\textbf{undef}, o_s)]\!]$

$\quad$ $(s_{guard}; \text{if } (\textbf{tobool}(e_{guard}))\ \{s_1\} \text{ else } \{s_2\},$

$\quad$ $\textbf{undef}, \overline{y_{guard}} \cup \bar{y_1} \cup \bar{y_2})$

9.20 Try/Catch/Finally

$\mathcal{T}[\![\texttt{try}\ s\ \texttt{catch}\ o_{\langle x,s\rangle}\ \texttt{fin}\ o_s]\!] =$

$\quad$ let $(s_{try}, \_, \overline{y_{try}}) = \mathcal{T}[\![s]\!]$

$\quad$ $\texttt{catchHandler} \in \mathit{JSVar} \times \mathit{JSStmt} \rightarrow \mathit{Variable} \times \mathit{Stmt} \times \overline{\mathit{Variable}}$

$\quad$ let $\texttt{catchHandler}(x, s') =$

$\quad\quad$ let $(s_{catch}, \_, \overline{y_{catch}}) = \mathcal{T}[\![s']\!]$

$\quad\quad$ $(\texttt{jsVarToNotJSVar}(x), s_{catch}, \overline{y_{catch}})$

$\quad$ let $\texttt{defaultCatchHandler}\ =$

$\quad\quad$ let $y = y$ is fresh

$\quad\quad$ $(y, \textbf{throw}\ y, \{y\})$

$\quad$ let $(x_{catch}, s_{catch}, \overline{y_{catch}}) =$

$$\begin{cases} \texttt{catchHandler}(x, s') & \text{if } o_{\langle x,s\rangle} = \textbf{some}\ (x, s') \\ \texttt{defaultCatchHandler} & \text{otherwise} \end{cases}$$

$\quad$ let $(s_{finally}, \_, \overline{y_{finally}}) = \texttt{getOrElse}(\textbf{undef}, o_s))$

$\quad$ $(\textbf{try-catch-fin}\ s_{try}\ x_{catch}\ s_{catch}\ s_{finally},$

$\quad$ $\textbf{undef}, \overline{y_{try}} \cup \overline{y_{catch}} \cup \overline{y_{finally}})$

9.21 Throw

$\mathcal{T}[\![\texttt{throw}\ e]\!] =$

$\quad$ let $(s, e', \bar{y}) = \mathcal{T}[\![e]\!]$

$\quad$ $(s; \texttt{throw}\ e', \textbf{undef}, \bar{y})$

9.22 Delete

The `Reference` type is specified by ECMA. In as few words as possible, a `Reference` corresponds to an object access, which takes the form $e_1.e_2$.

9.22.1 Delete `Reference`

$\mathcal{T}[\![\texttt{del } e_1.e_2]\!] =$

let $(s_{obj}, e_{obj}, \bar{y_1}) = \mathcal{T}[\![e_1]\!]$

let $(s_{field}, e_{field}, \bar{y_2}) = \mathcal{T}[\![e_2]\!]$

let $(s'_{field}, e'_{field}, \bar{y_3}) = \texttt{toString}(e_{field})$

let $y = y$ is fresh

$(s_{obj}; s_{field}; s'_{field};$

if $(e_{field} === \textbf{undef} \mid\mid e_{field} === \textbf{null})\{$

`throwTypeError`

$\}$ else $\{$

if $(\textbf{typeof}(e_{obj}) === \text{“}object''")\{$

$y := \texttt{del } e_{obj}.e'_{field}$

$\}$ else $\{$

$y := \textbf{true}$

$\}$

$\},$

$y, \bar{y_1} \cup \bar{y_2} \cup \bar{y_3} \cup \{y\})$

9.22.2 Delete Non-`Reference`

For this rule, a precondition is that $e \neq e_1.e_2$.

$\mathcal{T}[\![\texttt{del } e]\!] =$

let $(s, \_, \bar{y}) = \mathcal{T}[\![e]\!]$

$(s, \textbf{true}, \bar{y})$

10 notJS to notJS Passes

After the JavaScript → **notJS** pass is complete, **notJS** → **notJS** passes begin. Only relevant to the formalism is a single pass that adds in AST nodes to signal to the abstract interpreter to perform a join of states. In general, as many or as few **Merge** nodes can be inserted as desired without influencing correctness. However, this has a strong influence on precision and performance. In our use case, we insert **Merge** nodes at the following positions within a **notJS** AST:

1. Immediately after an entire conditional (`if`).
2. Immediately before the body of a `while` loop, and immediately after the whole `while` loop.
3. Immediately before the body of a `for` loop, and immediately after the whole `for` loop.
4. Typically, immediately after a labeled statement. For example: `lbl` : {`stmt`} **Merge**. The only exception to this is for function bodies, which are labeled with the special label ℓ_{return}. Merges here would prevent values from being returned from function calls correctly.
5. Immediately after function calls.
6. Immediately after **new** statements.
7. In `try`/`catch`/`finally`, immediately before the body of the `catch` clause, and immediately before the body of the `finally` clause.

11 Translator Optimizations

The formalism described above performs a correct translation, though the result is fairly naive. Specifically, the formalism does not show the presence of any optimizations. No optimizations were formalized for reasons of clarity and simplicity, though such optimizations have a significant impact on the size and performance of the translated programs. This section informally describes the optimizations which our translator implementation performs. These optimizations are described in no particular order.

11.1 `window`-specific Optimizations

Consider the JavaScript program below (whole program):

```
function foo() {
  x = 7;
  y = 2;
}
```

Because the variables `x` and `y` have not been declared with `var`, these are treated as properties of the global window object. This is handled in the **Make Global Variables `window` Properties** pass. This pass will translate the code above into something like the following:

```
function foo() {
  window."x" = 7;
  window."y" = 2;
}
```

Keep in mind that with this code, `window` refers to the program-inaccessible version of `window`.

While the above translation is correct, it is quite wasteful with respect to the $\mathcal{T}$ rules for object updates and object access. Specifically, $\mathcal{T}$ translates this to something like the following (simplified for illustration purposes):

```
function foo() {
  var temp1 = toObj(window);
  var temp2;
  if (isprim("x")) {
    temp2 = tostr("x");
  } else {
    temp2 = "x".toString();
  }
  temp1.temp2 = 7;
  var temp3 = toObj(window);
  var temp4;
  if (isprim("y")) {
    temp4 = tostr("y");
  } else {
    temp4 = "y".toString();
  }
  temp3.temp4 = 2;
}
```

With the above translation, there are three major problems as far as optimizations are concerned:

1. Given that `window` is inaccessible from the program, it can never be reassigned from the program. Additionally, the translation will never reassign `window` to any other value. As such, `window` is always guaranteed to be an object, so the use of `toObj` always acts as a no-op.
2. Any string constant is trivially a primitive. This means that both `else` branches in the above code are dead code. While concretely this is not a problem, abstractly this can become an issue if the semantics are imprecise. Theoretically, it is possible for an analysis to end up considering the dead code above as live, introducing spurious program paths and further harming precision.

3. String constrants are trivially strings. As such, the **tostr** operation in the code above always acts as a no-op, much in the same way as `toObj`.

Code like the above is introduced for any use of `window` properties, which can quickly cause AST bloat. As such, our translator implementation has specialized rules for dealing with object accesses and updates to `window` with constant strings. These rules safely bypass `toObj`, **isprim** and **tostr**, resulting in code that looks very similar to the input code.

11.2 Type-Specific Optimizations

As illustrated by the example in the previous subsection, the translation will often insert redundant conversions on constants. As such, our translator implementation is equipped with a basic type inferencer for **notJS** expressions (Exp). In contexts where the translation inserts dynamic checks which are based on types (i.e. **isprim**), the type inferencer is executed in an attempt to statically resolve the type. If the resolution is successful, then the emitting of dynamic checks can be bypassed. This same mechanism is also used to avoid inserting dynamic type conversions like **tostr** in contexts where they are no-ops.

11.3 Flattening Sequences

The translator will often produce heavily nested statements. For example, consider the following:

```
stmt1;
{
  stmt2;
  stmt3;
  {
    stmt4;
    stmt5;
  }
  stmt6;
}
```

In the above example, semantically there is no need for these statements to be nested in the way that they are. In other words, the following program is semantically identical to the example above:

```
stmt1;
stmt2;
stmt3;
stmt4;
stmt5;
stmt6;
```

Our translator implementation is equipped with a **notJS** $\rightarrow$ **notJS** pass that can perform this sort of sequence flattening.

While this may seem like a minor point, this does add a bit of a performance hit, is it means additional AST nodes must be processed. Moreover, this also simplifies two subsequent **notJS** $\rightarrow$ **notJS** optimization passes.

11.4 No-Op Elimination

During the translation process, no-op statements are frequently inserted using the `noOpStatement` macro. Oftentimes these statements end up being embedded within a sequence of statements. Within a sequence of statements, no-op statements are completely unnecessary. As such, a **notJS** $\rightarrow$ **notJS** pass exists to remove such statements from within a sequence. If the resulting sequence ends up being empty, as with a sequence of only no-op statements, then the whole sequence is replaced with a no-op statement. This process proceeds in a bottom-up fashion.

Removing such no-ops give a marginal performance increase. This also allows the next **notJS** $\rightarrow$ **notJS** optimization pass that will be described to be more effective.

11.5 Redundant Merge Node Elimination

After flattening sequences and removing no-op statements, it is not uncommon to encounter two or more **Merge** nodes in sequence. This is completely unnecessary in the abstract interpreter; performing a second merge immediately after performing a merge acts as an expensive no-op in the abstract world. As such, blocks of consecutive **Merge** nodes within a sequence are replaced with single **Merge** nodes.

11.6 Scratch Variables

The last optimization described is that of scratch variables. Scratch variables are always strongly updated, and have no equivalent in pure JavaScript. Just like regular variables, scratch variables are declared at the top of a given scope. These can be assigned to just like typical variables. The only restriction on scratch variables as far as the translation process is concerned is that scratch variables cannot be closed over. This limitation is relevant only in one case, illustrated below. Consider the following JavaScript:

```
var f = function foo() {
  ...
};
```

In this code, the variable `foo` must be made available within the function, and `foo` must be a handle on the function as a whole. As such, the translator translates this code into something like the following:

```
var temp;
var f = temp = function foo() {
   foo = temp;
   ...
};
```

In the above code, the function `foo` closes over `temp`, so `temp` cannot be a scratch variable. It is easy enough to introduce a new program variable here for `temp` instead.

Adding scratch variables resulted in massive gains in performance along with some gains in precision, as well. The only downside was that oftentimes many scratch variables were needed for even seemingly basic translations. It was not uncommon to see typical programs needing hundreds, if not thousands, of scratch variables in order to perform the translation.

To address this problem, an additional optimization was added. Oftentimes, the values of scratch variables do not need to cross JavaScript statement boundaries. This follows from the fact that scratch variables do not exist in pure JavaScript, so any sort of information flow between program statements in JavaScript must be performed via program variables. This property was exploited to allow for the reuse of scratch variables between JavaScript statements. To get a sense of the effect of this change, consider a sequence of JavaScript statements. Previously, the number of scratch variables needed for translation would equal the summation of the number of scratch variables needed to translate each individual statement within the sequence. By allowing for scratch variable reuse, summation becomes maximum; the number of scratch variables needed to translate a sequence of statements becomes the maximum number of scratch variables needed to translate each statement within. With this optimization, the number of scratch variables needed typically reduced by more than an order of magnitude.

The only case in which this aformentioned optimization added any difficulty was that of the `switch` statement. For `switch`, the temporary variables used for storing the result of the expression and whether ot not fllthrough behavior has been triggered must survive between JavaScript statements. This follows from the fact that individual cases can contain JavaScript statements. As such, program variables are used instead of scratch variables for `switch`.

It may also seem that loops are affected similarly to `switch`, given that the value of the guard is stored in a scratch variable and the body can contain JavaScript statements. However, for all loops with guards, it is never actually the case where the guard's value must survive a JavaScript statement. The guard's value is only needed for a brief period of time in between JavaScript statements, so scratch variables are appropriate in this context.

References

[1] Jslint. http://www.jslint.com/lint.html.

[2] Changhee Park, Hongki Lee, and Sukyoung Ryu. All about the with statement in javascript: Removing with statements in javascript applications. DLS '13, 2013.