1 Online Partial Evaluation

The bytecodes output by the specializer are not strictly the same as the ones output by the bytecode compiler. In particular, we add a `C_CALLN'` instruction whose syntax and semantics are different from the original `C_CALLN` instruction that it replaces.

1.1 Basic Stack Operations

$$\frac{m[cp] = \texttt{ACC}\ x}{\begin{array}{c}\langle \cdots S(st[x]) \cdots st, ac, cp, en, ea\rangle \rightarrow \\ \langle \cdots S(st[x]) \cdots st, S(st[x]), cp+2, en, ea\rangle \\ ow := !ow\end{array}}$$

We access a static value the normal way.

$$\frac{m[cp] = \texttt{ACC}\ x}{\begin{array}{c}\langle \cdots N(st[x]) \cdots st, ac, cp, en, ea\rangle \rightarrow \\ \langle \cdots N(st[x]) \cdots st, N(st[x]), cp+2, en, ea\rangle \\ ow := \texttt{ACC}(\mathsf{rf\_of\_}\ N(st[x])) :: !ow\end{array}}$$

A new `ACCESS` instruction is emitted when accessing a dynamic value. The access rank is the rank of the dynamic value amongst all the dynamic values in the stack at the time of the access.

$$\frac{m[cp] = \texttt{PUSH}}{\begin{array}{c}\langle st, S(ac), cp, en, ea\rangle \rightarrow \langle S(ac)\cdot st, S(ac), cp+1, en, ea\rangle \\ ow := !ow\end{array}}$$

We push static value the normal way.

$$\frac{m[cp] = \texttt{PUSH}}{\begin{array}{c}\langle st, N(ac), cp, en, ea\rangle \rightarrow \langle N(ac)\cdot st, N(ac), cp+1, en, ea\rangle \\ ow := \texttt{PUSH} :: !ow\end{array}}$$

Every time a dynamic value is pushed onto the stack, a `PUSH` instruction is emitted.

$$\frac{m[cp] = \texttt{POP 1}}{\begin{array}{c}\langle S(st[0]\cdot st, ac, cp, en, ea\rangle \rightarrow \langle st, ac, cp+2, en, ea\rangle \\ ow := !ow\end{array}}$$

We pop static value the normal way.

$$\frac{m[cp] = \texttt{POP 1}}{\begin{array}{c}\langle N(st[0])\cdot st, ac, cp, en, ea\rangle \rightarrow \langle st, ac, cp+2, en, ea\rangle \\ ow := \texttt{POP(1)}!ow\end{array}}$$

Every time we pop a dynamic value, we emit a `POP(1)` instruction.

$$\frac{m[cp] = \texttt{ASSIGN}\ n}{\begin{array}{c}\langle st[0]\cdots st[n-1]\cdot st[n]\cdot st, ac, cp, en, ea\rangle \rightarrow \\ \langle st[0]\cdots st[n-1]\cdot ac\cdot st, ac, cp+2, en, ea\rangle \\ ow := !ow\end{array}}$$

The `ASSIGN` instructions is used within loops. If the control parameters are known, there is no need to emit any code.

1.2 Access in Environment

TODO: Consider `ENVACC`

1.3 Function Application

$$\frac{m[cp] = \texttt{APPLY}\ x}{\begin{array}{c}\text{let } x = \mathsf{block}_{closure}(c, e) \text{ in} \\ \langle st[0]\cdots st[x-1]\cdot st, x, cp, en, ea\rangle \rightarrow \\ \langle st[0]\cdots st[x-1]\cdot cp+2\cdot en\cdot ea\cdot st, x, c, x, ea-1\rangle \\ ow := !ow\end{array}}$$

`APPLY` instructions are executed the normal way, except that some arguments maybe dynamic. Nothing is output.

$$\frac{m[cp] = \texttt{RETURN}\ k}{\begin{array}{c}\langle st[0]\cdots st[k-1]\cdot st, \mathsf{block}_{closure}(c,e), cp, en, ea\rangle \rightarrow \\ \langle st, \mathsf{block}_{closure}(c,e), c, \mathsf{block}_{closure}(c,e), ea-1\rangle \\ ow := \texttt{POP(}\mathsf{nb\_dyn\_in}\ k\texttt{)} ::!ow\end{array}}$$

We output as many as `POP` instructions as needed to pop the dynamic values that belong to the arguments of the function called.

TODO: Consider the two versions of the `RETURN` instruction. Consider `RESTART`, `GRAB`, `CLOSURE`, `CLOSUREREC`, and `OFFSETCLOSURE` in a dynamic context.

1.4 Access to Global Variables

TODO: Consider `GETGLOBAL` and `SETGLOBAL`.

1.5 Allocation of Blocks

TODO: Consider `MAKEBLOCK`.

1.6 Access to Components of Blocks

TODO: Consider `GETFIELD`.

1.7 Branches and Conditional Branches

TODO: Consider branches in dynamic contexts.

1.8 Calling C Functions

$$\frac{m[cp] = \texttt{C\_CALLN}\ 2\ \mathit{arraymake}}{\begin{array}{c}\langle S(\mathit{Const}(a_1))\cdot st, S(\mathit{Const}(a_0)), cp, en, ea\rangle \rightarrow \\ \langle st, S(\mathit{Pointer}(\mathsf{new\_block})) = \mathit{arraymake}(a_0,a_1), cp+2, en, ea\rangle \\ ow :=!ow\end{array}}$$

A call to the C function creating an array yields a pointer to a newly allocated block inside the heap. No output code.

$$\frac{m[cp] = \texttt{C\_CALLN}\ 2\ \mathit{arrayget}}{\begin{array}{c}\langle S(\mathit{Const}(a_1))\cdot st, S(\mathit{Pointer}(a_0)), cp, en, ea\rangle \rightarrow \\ \langle st, \mathit{arrayget}(a_0,a_1), cp+2, en, ea\rangle \\ ow :=!ow\end{array}}$$

No outout code.

$$\frac{m[cp] = \texttt{C\_CALLN}\ 2\ \mathit{arrayget}}{\begin{array}{c}\langle S(\mathit{Const}(a_1))\cdot st, N(\mathit{Unk}), cp, en, ea\rangle \rightarrow \\ \langle st, N(\mathit{Unk}), cp+2, en, ea\rangle \\ ow := \texttt{POP(1)} :: \texttt{C\_CALLN'(2,}\mathit{arrayget}\texttt{)} :: \texttt{ACC(1)} :: \texttt{PUSH} \\ :: \texttt{CONST(}a_1\texttt{)} :: \texttt{PUSH} ::!ow\end{array}}$$

The call to `C_CALLN` has been changed into a call to `C_CALLN'`. The first actually resolves the relocation, the second one will access directly to the relocated resource. This is necessary since the relocation depends upon the absolute position of the instruction in the block of bytecodes. In the specialized chunk of code, this is not valid anymore.

$$\frac{m[cp] = \texttt{C\_CALLN}\ 2\ \mathit{arrayget}}{\begin{array}{c}\langle S(\mathit{Const}(a_1))\cdot st, N(\mathit{Pointer}(a_0)), cp, en, ea\rangle \rightarrow \\ \langle st, N(\mathit{Unk}), cp+2, en, ea\rangle \\ ow := \texttt{C\_CALLN'(2,}\mathit{arrayget}\texttt{)} :: \texttt{CONST(}a_0\texttt{)} :: \texttt{PUSH} \\ :: \texttt{CONST(}a_1\texttt{)} ::!ow\end{array}}$$

Here, the pointer is known, but the contents are dynamic. We do not generate the same instruction as above.

$$\frac{m[cp] = \texttt{C\_CALLN}\ 3\ \mathit{arrayset}}{\begin{array}{c}\langle S(\mathit{Const}(a_1))\cdot N(\mathit{Unk})\cdot st, S(\mathit{Pointer}(a_0)), cp, en, ea\rangle \rightarrow \\ \langle st, S(\mathit{Pointer}(a_0)), cp+2, en, ea\rangle \\ ow := \texttt{C\_CALLN'(3,}\mathit{arrayset}\texttt{)} :: \texttt{CONST(}a_0\texttt{)} :: \texttt{PUSH} \\ :: \texttt{CONST(}a_1\texttt{)} ::!ow\end{array}}$$

Things to notice. The pointer is still static in the sense it still points to the same cell, but the contents of the cell have been updated with a dynamic value. The output code is not 'well-typed' anymore. The `C_CALLN'` instruction accepts both pointers and constants as pointer designations.

1.9 Integer Constants

$$\frac{m[cp] = \texttt{CONST}\ x}{\langle st, ac, cp, en, ea\rangle \rightarrow \langle st, S(\mathit{Const}(x)), cp+1, en, ea\rangle}$$
$$ow := !ow$$

`CONST` instructions create static constant values.

1.10 Integer Arithmetic

$$\frac{m[cp] = \texttt{ADDINT}}{\begin{array}{c}\langle S(\mathit{Const}(a_1)) \cdot st, S(\mathit{Const}(a_0)), cp, en, ea\rangle \rightarrow \\ \langle st, S(\mathit{Const}(a_0 + a_1)), cp+1, en, ea\rangle\end{array}}$$
$$ow := !ow$$

$$\frac{m[cp] = \texttt{ADDINT}}{\langle N(\mathit{Unk}) \cdot st, S(\mathit{Const}(ac)), cp, en, ea\rangle \rightarrow \langle st, N(\mathit{Unk}), cp+1, en, ea\rangle}$$
$$ow := \texttt{ADDINT} :: \texttt{CONST}(ac) :: !ow$$

$$\frac{m[cp] = \texttt{ADDINT}}{\langle S(\mathit{Const}(a_0)) \cdot st, N(\mathit{Unk}), cp, en, ea\rangle \rightarrow \langle st, N(\mathit{Unk}), cp+1, en, ea\rangle}$$
$$ow := \texttt{POP(1)} :: \texttt{ADDINT} :: \texttt{ACC(1)} :: \texttt{PUSH} :: \texttt{CONST}(a_0) :: \texttt{PUSH} :: !ow$$

$$\frac{m[cp] = \texttt{ADDINT}}{\langle N(\mathit{Unk}) \cdot st, N(\mathit{Unk}), cp, en, ea\rangle \rightarrow \langle st, N(\mathit{Unk}), cp+1, en, ea\rangle}$$
$$ow := \texttt{ADDINT} :: !ow$$

$$\frac{m[cp] = \texttt{MULINT}}{\begin{array}{c}\langle S(\mathit{Const}(a_1)) \cdot st, S(\mathit{Const}(a_0)), cp, en, ea\rangle \rightarrow \\ \langle st, S(\mathit{Const}(a_0 \times a_1)), cp+1, en, ea\rangle\end{array}}$$
$$ow := !ow$$

$$\frac{m[cp] = \texttt{MULINT}}{\langle N(\mathit{Unk}) \cdot st, S(\mathit{Const}(ac)), cp, en, ea\rangle \rightarrow \langle st, N(\mathit{Unk}), cp+1, en, ea\rangle}$$
$$ow := \texttt{MULINT} :: \texttt{CONST}(ac) :: !ow$$

$$\frac{m[cp] = \texttt{MULINT}}{\langle S(\mathit{Const}(a_0)) \cdot st, N(\mathit{Unk}), cp, en, ea\rangle \rightarrow \langle st, N(\mathit{Unk}), cp+1, en, ea\rangle}$$
$$ow := \texttt{POP(1)} :: \texttt{MULINT} :: \texttt{ACC(1)} :: \texttt{PUSH} :: \texttt{CONST}(a_0) :: \texttt{PUSH} :: !ow$$

$$\frac{m[cp] = \texttt{MULINT}}{\langle N(\mathit{Unk}) \cdot st, N(\mathit{Unk}), cp, en, ea\rangle \rightarrow \langle st, N(\mathit{Unk}), cp+1, en, ea\rangle}$$
$$ow := \texttt{MULINT} :: !ow$$

TODO: `EQ`, `SUBINT`