1 TODO

- Linking
- Component
- Cross-language LR

2 The two capability machines

2.1 Domains

Source language domains:

$$
\begin{array}{rrcl}
\mathrm{a}, \mathrm{base} \in & \mathrm{Addr} & \overset{def}{=} & \mathbb{N} \\
 & \mathrm{Word} & \overset{def}{=} & \mathbb{Z} \uplus \mathrm{Cap} \\
\mathrm{perm} \in & \mathrm{Perm} & ::= & \ldots \\
\mathrm{g} \in & \mathrm{Global} & ::= & \mathrm{global} \mid \mathrm{local} \\
 & \mathrm{Cap} & ::= & \{((\mathrm{perm}, \mathrm{g}), \mathrm{base}, \mathrm{end}, \mathrm{a}) \mid \mathrm{end} \in \mathrm{Addr} \cup \{\infty\}\} \uplus \\
 & & & \{\text{stack-ptr}(\mathrm{perm}, \mathrm{base}, \mathrm{end}, \mathrm{a})\} \uplus \\
 & & & \{\text{ret-ptr}\} \\
 & \mathrm{RegisterName} & ::= & \mathrm{pc} \mid \mathrm{r_{ret}} \mid \mathrm{r_{stk}} \mid \ldots \\
 & \mathrm{RegisterFile} & \overset{def}{=} & \mathrm{RegisterName} \rightarrow \mathrm{Word} \\
 & \mathrm{Memory} & \overset{def}{=} & \mathrm{Addr} \rightarrow \mathrm{Word} \\
 & \mathrm{MemorySegment} & \overset{def}{=} & \mathrm{Addr} \rightharpoonup \mathrm{Word} \\
\mathrm{frame} \in & \mathrm{StackFrame} & \overset{def}{=} & \mathrm{Cap} \times \mathrm{MemorySegment} \times \mathrm{Addr} \\
\mathrm{stk} \in & \mathrm{Stack} & \overset{def}{=} & \mathrm{StackFrame}^* \\
\Phi \in & \mathrm{ExecConf} & \overset{def}{=} & \mathrm{Memory} \times \mathrm{RegisterFile} \times \mathrm{Stack} \times \mathrm{MemorySegment} \\
 & \mathrm{Conf} & \overset{def}{=} & \mathrm{ExecConf} \uplus \{\mathrm{failed}\} \uplus (\{\mathrm{halted}\} \times \mathrm{Memory})
\end{array}
$$

The target language domains are all the non blue parts in the above.

More convenient definitions

$$\Phi(r) \overset{def}{=} \Phi.\mathrm{reg}(r)$$

where $r \in \mathrm{RegisterName}$

2.2 Syntax

The target machine is[dashed] a simple capability machine with memory capabilities, local capabilities and enter capabilities. The syntax of the instructions of the target machine is defined as follows:

$$
\begin{array}{rcl}
n & \in & \mathbb{N} \\
r & \in & \mathrm{RegisterName} \\
rn & ::= & r \mid n \\
i & ::= & \mathit{fail} \mid \mathit{halt} \mid \mathit{jmp}\ r \mid \mathit{jnz}\ r\ rn \mid \mathit{isptr}\ r\ r \mid \mathit{geta}\ r\ r \mid \mathit{getb}\ r\ r \mid \\
 & & \mathit{gete}\ r\ r \mid \mathit{getp}\ r\ r \mid \mathit{getl}\ r\ r \mid \mathit{move}\ r\ rn \mid \mathit{store}\ r\ r \mid \\
 & & \mathit{load}\ r\ r \mid \mathit{lea}\ r\ rn \mid \mathit{restrict}\ r\ r\ rn \mid \mathit{subseg}\ r\ rn\ rn \mid \\
 & & \mathit{plus}\ r\ rn\ rn \mid \mathit{minus}\ r\ rn\ rn \mid \mathit{seta2b}\ r
\end{array}
$$

The source machine is also a capability machine with memory capabilities, local capabilities and enter capabilities. Unlike the target machine, the source machine is going to have a built in stack. The syntax of the source machine language is as follows:

$$\begin{array}{rcl} n & \in & \mathbb{N} \\ r & \in & \mathrm{RegisterName} \setminus \{\mathrm{r_{ret}}, \mathrm{r_{stk}}\} \\ rn & ::= & r \mid n \\ i & ::= & i \mid call\ r\ rn \end{array}$$

There is one suntactic difference between the source language and the target language, namely the target language has an extra instruction in the *call* instruction.

2.3 Permissions

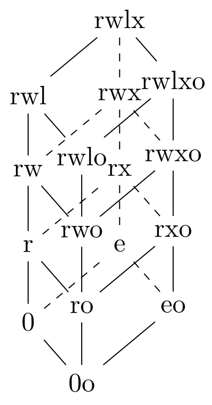

Figure 1: Permission hierarchy

2.4 Operational Semantics

2.4.1 Notes

Genrally:

- Opaque capabilities: Only allow locality and permission to be inspected (i.e., no addresses disclosed).

Source language:

- No jumps to enter (call reserved for this)
- Variable length instructions that match the length of the compiled instructions
  - Opaque capabilities do not hide the length of a program (especially an issue if we ever hope to have error recovery).
  - This is needed for correctness. See subsection 3.1 for an example it helps prevent.

Target language:

-

2.4.2 Helpful functions

$$updatePc(\stackrel{def}{=})\dots$$

$$updatePcPerm(\stackrel{def}{=})\dots$$

$$clear_{ms}(\stackrel{def}{=})\dots$$

$$clear_{loc,reg}(\stackrel{def}{=})\dots$$

2.4.3 Source Language

$$[\![jmp\ r]\!]_{\mathrm{src}}(\Phi) = \begin{cases} \Phi\,[\mathrm{reg.pc} \mapsto opc] & \Phi(r) = \text{ret-ptr and } \Phi.\mathrm{stk} = (opc, ms'_{stk}, a'_{stk}) :: stk' \\ \quad [\mathrm{ms_{stk}} \mapsto ms'_{stk} \uplus clear_{ms}(ms_{stk})] & \\ \quad [\mathrm{a_{stk}} \mapsto a'_{stk}] & \\ \quad [\mathrm{stk} \mapsto stk'] & \\ & \\ \Phi[\mathrm{reg.pc} \mapsto updatePcPerm(\Phi(r))] & \text{otherwise} \end{cases}$$

$$[\![call\ r\ rn]\!]_{\mathrm{src}}(\Phi) = \begin{cases} (mem, reg', stk', ms'_{stk}) & \Phi = (mem, reg, stk, ms_{stk}) \text{ and} \\ & \\ \text{failed} & \text{otherwise} \end{cases}$$

2.4.4 Target Language

2.5 Compiler

$$[\cdot] : i^* \to i^*$$

3 Examples

3.1 Capability Opacity

This example was introduced when we envisioned a capability machine with *push*, *pop*, *sload*, *call* and *return* instructions. The below example is a motivation for having variable length instructions in the source language because if we have enough memory to "do the same" in the target language as in the source language, then the below example does not work.

The following pseudo program demonstrates the need of opaque capabilities. If we assume a system with no opaque capabilities, then the following programs break compiler correctness

```
p1 ::= if r1 is length 2 then
         call r1 with the following callback in r5:
           {put diverging closure in r2;
            return};
         halt
       else diverge

p2 ::= if r1 is length 2 then
         call r1 with the following callback in r5:
           {put terminating closure in r2;
            return};
         halt
       else diverge
```

The diverging closure could just contain *jmp* pc, which diverges. The terminating closure could just be *return*.

The context with $r1$ as an executable capability pointing to:

call $r5$ 0
jmp $r5$

distinguishes the two contexts, but two instructions are not enough to do the same at the target level. We would not have enough instructions to set up a proper return pointer for the compiled return to use.

4 Back translation

We want to do the back translation by interpreting the programs.