Concrete Semantics for CrabIR

- A reference is an address (integer) within a region
- A cell $\langle b, o \rangle$ is the data associated to a reference
  - b is the base address and o is a numerical offset
  - Represent an integer o (if $b = 0$) or another reference $b + o$ (if $b \neq 0$)
- Program state σ=($numEnv$, $refEnv$, $rgnEnv$, $memObjs$, $rgnAddrs$)
  1. $numEnv \in (\mathcal{V}_{\mathcal{B}} \cup \mathcal{V}_{\mathcal{I}}) \mapsto \mathbb{B} \cup \mathbb{Z}$
  2. $refEnv \in \mathcal{V}_{\mathcal{R}ef} \mapsto \text{Cell}$
  3. $rgnEnv \in \mathcal{V}_{\mathcal{R}gn} \mapsto (\text{Address} \mapsto \text{Cell})$
  4. $memObjs \in \text{List}(\text{Address} \times \text{Address})$
  5. $rgnAddrs \in \mathcal{V}_{\mathcal{R}gn} \mapsto 2^{\text{Address}}$
- The semantics for statements is given by $[\![.]\!]_\Omega(.)$:
  $[\![.]\!]_\Omega(.) : S \mapsto \sigma_\Omega \mapsto \sigma_\Omega$ $\quad$ ($\sigma_\Omega = \sigma \cup \{\Omega\}$ and Ω denoting error)

$$[\![stmt]\!]_\Omega(\sigma) = \begin{cases} \Omega & \text{if } \sigma = \Omega \\ [\![stmt]\!](\sigma) & \text{otherwse} \end{cases}$$

Conversion Between Cells and Addresses

cell2Addr(c)

$\quad$ `match` c `with` $\langle base, o\rangle \ \rightarrow base + o$

addr2Cell(a, *blkList*)

$\quad$ `let` $\langle start, end\rangle = \text{find}(a, blkList)$ `in`

$\quad$ `if` $start = end = 0$ `then` $\langle 0, 0\rangle$ `else` $\langle start, a - start\rangle$

find(a, *blkList*)

$\quad$ `match` *blkList* `with`

$\quad\quad$ nil $\rightarrow \ \langle 0, 0\rangle$

$\quad\quad \langle start, end\rangle :: tail \ \rightarrow$

$\quad\quad\quad$ `if` $(start \leq a$ `and` $\ a < end)$ `then` $\langle start, end\rangle$

$\quad\quad\quad$ `else` find(a, *tail*)

member(a, *blkList*)

$\quad \text{find}(a, blkList) \neq \langle 0, 0\rangle$

Generate the Next Available Address

```
nextAddr(memObjs)
    match memObjs with
        nil  →  freshAddress()
        ⟨_, end⟩ :: nil  →  end
        _ :: tail  →  nextAddr(tail)
```

- Addresses are generated in increasing order
- freshAddress() returns an address divisible by the word size

$[\![Stmt_{rgn}]\!](\sigma)$

$[\![\texttt{initrgn}(rgn)]\!](\sigma)$

$$
\begin{array}{l}
\quad \texttt{match } \sigma \texttt{ with } (numEnv, refEnv, rgnEnv, memObjs, rgnAddrs) \ \rightarrow \\
\quad \texttt{if } rgn \in \mathrm{dom}(rgnAddrs) \texttt{ then } \sigma \\
\quad \texttt{else} \\
\qquad \texttt{let } rgnAddrs' = rgnAddrs[rgn \mapsto \emptyset] \texttt{ in} \\
\qquad (numEnv, refEnv, rgnEnv, memObjs, rgnAddrs')
\end{array}
$$

$[\![Stmt_{rgn}]\!](\sigma)$

$[\![rgn' := \texttt{copyrgn}(rgn)]\!](\sigma)$

$$
\begin{array}{l}
\texttt{match } \sigma \texttt{ with } (numEnv, refEnv, rgnEnv, memObjs, rgnAddrs) \ \rightarrow \\
\texttt{if } rgn \notin \mathrm{dom}(rgnAddrs) \texttt{ or } rgn' \notin \mathrm{dom}(rgnAddrs) \texttt{ then } \Omega \\
\texttt{else} \\
\quad \texttt{let } rgnAddrs' = rgnAddrs[rgn' \mapsto rgnAddrs(rgn)] \texttt{ in} \\
\quad \texttt{let } rgnEnv' = \texttt{if } rgn \in \mathrm{dom}(rgnEnv) \\
\qquad\quad \texttt{then } rgnEnv[rgn' \mapsto rgnEnv(rgn)] \\
\qquad\quad \texttt{else } rgnEnv \texttt{ in} \\
\quad (numEnv, refEnv, rgnEnv', memObjs, rgnAddrs')
\end{array}
$$

- Useful for purification of functions at encoding time: make a copy of a region to remember the region's contents before the execution of a function's body

$[\![Stmt_{rgn}]\!](\sigma)$

$[\![ref := \texttt{makeref}(rgn, n)]\!](\sigma)$

$$
\begin{array}{l}
\texttt{match } \sigma \texttt{ with } (numEnv, refEnv, rgnEnv, memObjs, rgnAddrs) \ \rightarrow \\
\texttt{if } rgn \notin \mathrm{dom}(rgnAddrs) \texttt{ or } \ [\![n]\!](\sigma) \leq 0 \texttt{ then } \Omega \\
\texttt{else} \\
\quad \texttt{let } base := \mathrm{nextAddr}(memObjs) \texttt{ in} \\
\quad \texttt{let } sz := [\![n]\!](\sigma) \texttt{ in} \\
\quad \texttt{let } refEnv' = refEnv[ref \mapsto \langle base, 0 \rangle] \texttt{ in} \\
\quad \texttt{let } memObjs' = \langle base, base + sz \rangle :: memObjs \texttt{ in} \\
\quad \texttt{let } rgnAddrs' = rgnAddrs[rgn \mapsto rgnAddrs(rgn) \cup \{base\}] \texttt{ in} \\
\quad (numEnv, refEnv', rgnEnv, memObjs', rgnAddrs')
\end{array}
$$

- Valid range of addresses $[lb, ub]$ for ref can be enforced:

$$ref := \texttt{makeref}(rgn, n), \texttt{assume}(ref \geq lb), \texttt{assume}(ref \leq ub)$$

$[\![Stmt_{rgn}]\!](\sigma)$

$$
\begin{array}{l}
[\![(rgn_2, ref_2) := \texttt{gepref}(rgn_1, ref_1, n)]\!](\sigma) \\
\quad \texttt{match } \sigma \texttt{ with } (numEnv, refEnv, rgnEnv, memObjs, rgnAddrs) \ \rightarrow \\
\quad \texttt{if } \{rgn_1, rgn_2\} \not\subseteq \mathrm{dom}(rgnAddrs) \texttt{ or } ref_1 \notin \mathrm{dom}(refEnv) \texttt{ then } \Omega \\
\quad \texttt{else} \\
\qquad \texttt{let } \langle b, o \rangle = refEnv(ref_1) \texttt{ in} \\
\qquad \texttt{let } o' = o + [\![n]\!](\sigma) \texttt{ in} \\
\qquad \texttt{if } \mathrm{find}(b + o, memObjs) \neq \mathrm{find}(b + o', memObjs) \texttt{ then } \Omega \\
\qquad \texttt{else} \\
\qquad\quad \texttt{let } refEnv' = refEnv[ref_2 \mapsto \langle b, o' \rangle] \texttt{ in} \\
\qquad\quad \texttt{let } rgnAddrs' = rgnAddrs[rgn_2 \mapsto rgnAddrs(rgn_2) \cup \{b + o'\}] \texttt{ in} \\
\qquad\quad (numEnv, refEnv', rgnEnv, memObjs, rgnAddrs')
\end{array}
$$

- Model pointer arithmetic
- Allow to "switch" between regions but within same memory block

$[\![Stmt_{rgn}]\!](\sigma)$

$[\![lhs := \texttt{loadref}(rgn, ref)]\!](\sigma)$

$$
\begin{array}{l}
\texttt{match } \sigma \texttt{ with } (numEnv, refEnv, rgnEnv, memObjs, rgnAddrs) \ \rightarrow \\
\texttt{if } rgn \notin \mathrm{dom}(rgnEnv) \texttt{ or } \ ref \notin \mathrm{dom}(refEnv) \texttt{ then } \Omega \\
\texttt{else} \\
\quad \texttt{let } a = \mathrm{cell2Addr}(refEnv(ref)) \texttt{ in} \\
\quad \texttt{let } \mathcal{M} = rgnEnv(rgn) \texttt{ in} \\
\quad \texttt{if } \neg \ \mathrm{member}(a, memObjs) \texttt{ or } a \notin \mathrm{dom}(M) \texttt{ then } \Omega \\
\quad \texttt{else} \\
\quad\quad \texttt{let } (numEnv', refEnv') = \\
\quad\quad \begin{cases} (numEnv, refEnv[lhs \mapsto \mathcal{M}(a)]) & \text{if } \mathrm{type}(lhs) = \textbf{ref} \\ (numEnv[lhs \mapsto \mathrm{cell2Addr}(\mathcal{M}(a))], refEnv) & otherwise \end{cases} \\
\quad\quad \texttt{in} \\
\quad\quad (numEnv', refEnv', rgnEnv, memObjs, rgnAddrs)
\end{array}
$$

$[\![Stmt_{rgn}]\!](\sigma)$

$[\![\texttt{storeref}(rgn, ref, val)]\!](\sigma)$

$$
\begin{array}{l}
\texttt{match } \sigma \texttt{ with } (numEnv, refEnv, rgnEnv, memObjs, rgnAddrs) \ \rightarrow \\
\texttt{if } rgn \notin \mathrm{dom}(rgnAddrs) \texttt{ or } \ ref \notin \mathrm{dom}(refEnv) \ \texttt{ or} \\
\quad (\mathrm{type}(val) \neq \textbf{ref} \texttt{ or } val \notin \mathrm{dom}(refEnv)) \texttt{ then } \Omega \\
\texttt{else} \\
\quad \texttt{let } a = \mathrm{cell2Addr}(refEnv(ref)) \texttt{ in} \\
\quad \texttt{if } \neg \ \mathrm{member}(a, memObjs) \texttt{ then } \Omega \\
\quad \texttt{else} \\
\qquad \texttt{let } c = \begin{cases} refEnv(val) & \text{if } \mathrm{type}(val) = \textbf{ref} \\ \langle 0, [\![val]\!](\sigma)\rangle & \text{otherwise} \end{cases} \\
\qquad \texttt{in} \\
\qquad \texttt{let } \mathcal{M} = rgnEnv(rgn) \texttt{ in} \\
\qquad \texttt{let } rgnEnv' = rgnEnv[rgn \mapsto \mathcal{M}[a \mapsto c]] \texttt{ in} \\
\qquad (numEnv, refEnv, rgnEnv', memObjs, rgnAddrs)
\end{array}
$$

$[\![Stmt_{rgn}]\!](\sigma)$

$[\![x := \texttt{reftoint}(rgn, ref)]\!](\sigma)$

$\texttt{match } \sigma \texttt{ with } (numEnv, refEnv, rgnEnv, memObjs, rgnAddrs) \rightarrow$
$\texttt{if } rgn \notin \text{dom}(rgnAddrs) \texttt{ or } ref \notin \text{dom}(refEnv)) \texttt{ then } \Omega$
$\texttt{else}$
$\quad \texttt{let } numEnv' = numEnv[x \mapsto \text{cell2Addr}(refEnv(ref))] \texttt{ in}$
$\quad (numEnv', refEnv, rgnEnv, memObjs, rgnAddrs)$

$[\![Stmt_{rgn}]\!](\sigma)$

$[\![ref := \texttt{inttoref}(rgn, x)]\!](\sigma)$

$\texttt{match } \sigma \texttt{ with } (numEnv, refEnv, rgnEnv, memObjs, rgnAddrs) \rightarrow$
$\texttt{if } rgn \notin \text{dom}(rgnAddrs) \texttt{ then } \Omega$
$\texttt{else}$
$\quad \texttt{let } c = \text{addr2Cell}([\![x]\!](\sigma), memObjs) \texttt{ in}$
$\quad \texttt{if } c = \langle 0, 0 \rangle \texttt{ then } \Omega$
$\quad \texttt{else}$
$\quad\quad \texttt{let } refEnv' = refEnv[ref \mapsto c] \texttt{ in}$
$\quad\quad \texttt{let } rgnAddrs' = rgnAddrs[rgn \mapsto rgnAddrs(rgn) \cup \{[\![x]\!](\sigma)\}] \texttt{ in}$
$\quad\quad (numEnv, refEnv', rgnEnv, memObjs, rgnAddrs')$

Well-Formed CrabIR Programs

- There are some conditions for a Crab program to "make sense"
- A strongly-typed system is a major part of it but not all
- All regions must be initialized before they can be used:
  - `initrgn`(*rgn*) must dominate any use of *rgn*
- All references must be initialized before they can be used. A reference *ref* can be only initialized by
  - $ref := \texttt{makeref}(rgn, n)$,
  - $(rgn, ref) := \texttt{gepref}(\ldots)$, or
  - $ref := \texttt{inttoref}(rgn, \ldots)$

Writing CrabIR Programs

- Users can write CrabIR programs but we do not expect it
- CrabIR has been designed to perform analysis of LLVM bitcode
- Translation from LLVM bitcode to CrabIR should:
  - Perform whole-program pointer analysis
  - Perform function purification: use pointer analysis to eliminate side-effects
  - Partition memory into regions: identify for each LLVM memory instruction its corresponding regions
- Available open-source translator from LLVM to CrabIR: `https://github.com/seahorn/crab-llvm`
- Available open-source whole-pointer analysis for LLVM: `https://github.com/seahorn/sea-dsa`