

Assignment #5

Diego Oniarti - 257835

1 Encoding ULC into System F using recursive types

Try to define type τ_u , which is the type that any ULC term can be given in F+isorecursive types. If you can define τ_u , define a function inductively on ULC terms so that it maps any ULC term to a term of F+isorecursive types whose type is τ_u and that has the same behaviour as the original ULC term (i.e., if you map an ULC application, you get something that eventually behaves like an application).

If you cannot define τ_u , argue why it cannot exist.

In this case, consider ULC terms to be: $t ::= n|x|\lambda x.t|t|\langle t, t \rangle|t.1|t.2|inl\ t|inr\ t|case\ t\ of\ \begin{cases} inl\ x_1 \rightarrow t \\ inr\ x_2 \rightarrow t \end{cases}$

Encoding these terms into lambdas is not an option.

$$\tau_u \triangleq \mu\alpha.(\mathbb{N} \uplus (\alpha \rightarrow \alpha)) \uplus ((\alpha \uplus \alpha) \uplus (\alpha \times \alpha))$$

Base cases

$$\begin{aligned} F(n) &= \text{fold}_{\mu_U}(\text{inl } (\text{inl } n)) \\ F(\lambda x.t) &= \text{fold}_{\mu_U}(\text{inl } (\text{inr } \lambda x : \mu_U.F(t))) \\ F(\langle t_1, t_2 \rangle) &= \text{fold}_{\mu_U}(\text{inr } (\text{inr } (F(t_1), F(t_2)))) \\ F(\text{inl } t) &= \text{fold}_{\mu_U}(\text{inr } (\text{inl } (\text{inl } (F(t))))) \\ F(\text{inr } t) &= \text{fold}_{\mu_U}(\text{inr } (\text{inl } (\text{inr } (F(t))))) \end{aligned}$$

Inductive cases

$$\begin{aligned} F(t_1 t_2) &= \text{case unfold}(F(t_1)) \text{ of } \begin{cases} \text{inl } x_1 \mapsto \text{case } x_1 \text{ of } \begin{cases} \text{inl } x_{11} \mapsto \Omega \\ \text{inr } x_{12} \mapsto x_{12} F(t_2) \end{cases} \\ \text{inr } x_2 \mapsto \Omega \end{cases} \\ F(t.1) &= \text{case unfold}(F(t)) \text{ of } \begin{cases} \text{inl } x_1 \mapsto \Omega \\ \text{inr } x_2 \mapsto \text{case } x_2 \text{ of } \begin{cases} \text{inl } x_{21} \mapsto \Omega \\ \text{inr } x_{22} \mapsto x_{22.1} \end{cases} \end{cases} \\ F(t.2) &= \text{case unfold}(F(t)) \text{ of } \begin{cases} \text{inl } x_1 \mapsto \Omega \\ \text{inr } x_2 \mapsto \text{case } x_2 \text{ of } \begin{cases} \text{inl } x_{21} \mapsto \Omega \\ \text{inr } x_{22} \mapsto x_{22.2} \end{cases} \end{cases} \\ F\left(\text{case } t_0 \text{ of } \begin{cases} \text{inl } x_1 \mapsto t_1 \\ \text{inr } x_2 \mapsto t_2 \end{cases}\right) &= \text{case unfold}(F(t_0)) \text{ of } \begin{cases} \text{inr } x_2 \mapsto \text{case } x_2 \text{ of } \begin{cases} \text{inl } x_{21} \mapsto \text{case } x_{21} \text{ of } \begin{cases} \text{inl } x_{211} \mapsto F(t_1) \\ \text{inr } x_{212} \mapsto F(t_2) \end{cases} \\ \text{inr } x_{22} \mapsto \Omega \end{cases} \end{cases} \end{aligned}$$

2 Formalising ASM capability machines

Capability machines extend assembly instructions with explicit capabilities such that reading and writing on memory is only allowed if a capability is provided.

Take ASM without the private heap and extend the language with capabilities and formalise their semantics: call this CASM. You choose how to model them, choose wisely according to their behaviour as described below.

Capabilities are unforgeable and unobservable tokens which the program can create. At the start, each memory location is unprotected. The language must provide instructions for creating a capability, and for protecting a location given a capability, this should only be possible if the location is unprotected.

Reading and writing a memory location is always possible if the location is unprotected. However, if the location is protected with a capability, reading and writing that location is only possible if the same capability is provided at reading and writing time.

Capabilities are gonna be represented as negative numbers. To accomodate this the memory is no longer a mapping from \mathbb{N} to \mathbb{N} but one from \mathbb{N} to \mathbb{I} .

To make the capabilities unforgeable and unobservable, registers can still only hold positive values. Similarly the rest of the operations are only allowed to operate on positive numbers.

$$\begin{aligned}
 t ::= & r := n \\
 & | \text{sum } r r \\
 & | \text{sub } r r \\
 & | \text{cmp } r r \\
 & | \text{jmp } r \\
 & | \text{jiz } r \\
 & | \text{jeq } r \\
 & | \text{read } r r \\
 & | \text{write } r r \\
 & | \text{cap } r \\
 & | \text{prot } r \\
 & | \text{deadp } r \\
 & | \text{writep } r \\
 r ::= & ar | br | cr | dr | er | fr | gr | hr \\
 & | ir | jr | kr | lr | mr | nr | or \\
 C ::= & \emptyset | C, \mathbb{N} \mapsto t \\
 F ::= & \emptyset | F, \mathbb{N} \mapsto b \\
 R ::= & \emptyset | R, r \mapsto \mathbb{N} \\
 M ::= & \emptyset | M, \mathbb{N} \mapsto \mathbb{I} \\
 P ::= & \emptyset | P, \mathbb{I} \mapsto \mathbb{N}
 \end{aligned}$$

Judgement:

$$n; C; R; F; M; P \Rightarrow n; C; R; F; M; P$$

Codebase, registers, flags, memory, and protections:

$$\begin{array}{c}
 \frac{C = C', n \mapsto t}{C(n) = t} \quad \frac{R = R', r \mapsto n}{R(r) = n} \quad \frac{F = F', n \mapsto B}{F(n) = B} \\
 \frac{C = C', n' \mapsto - \quad C'(n) = t}{C(n) = t} \quad \frac{R = R', r' \mapsto - \quad R'(r) = n}{R(r) = n} \quad \frac{F = F', n' \mapsto - \quad F'(n) = B}{F(n) = B} \\
 \frac{M = M', n \mapsto n'}{M(n) = n'} \quad \frac{M = M', n'' \mapsto - \quad M'(n) = n'}{M(n) = n'} \\
 \frac{P = P', i \mapsto n}{P(i) = n} \quad \frac{P = P', i' \mapsto - \quad P'(i) = n}{P(i) = n} \quad \frac{P = P', i' \mapsto n' \quad P'(i) = n}{P(i) = n}
 \end{array}$$

The rule for traversing the protection list is a bit different from the others because it has to allow the same capability to

be mapped to multiple memory addresses.

Capabilities will be generated in a sequential order starting from -1 . The function FREE returns the first unused negative number

$$\begin{aligned}\text{FREE}(P) &= \text{MINP}(P) - 1 \\ \text{MINP}(\emptyset) &= 0 \\ \text{MINP}(P, i \mapsto n) &= \min(i, \text{MINP}(P))\end{aligned}$$

The UNPROT function checks if a memory address is unprotected

$$\begin{aligned}\text{UNPROT}(\emptyset, n) &= \text{true} \\ \text{UNPROT}(P, i \rightarrow n, n) &= \text{false} \\ \text{UNPROT}(P, i \rightarrow n', n) &= \text{UNPROT}(P)\end{aligned}$$

Rules:

$$\frac{C(n) = r_1 := n_1 \quad R' = R, r_1 \mapsto n_1 \quad n' = n + 1}{n; C; R; F; M; P \Rightarrow n'; C; R'; F; M; P} \text{load}$$

$$\frac{C(n) = \text{sum } r_1 \ r_2 \ R(r_1) = n_1 \ R(r_2) = n_2 \ R' = R, r_1 \mapsto n_1 + n_2 \ n' = n + 1}{n; C; R; F; M; P \Rightarrow n'; C; R'; F; M; P} \text{sum}$$

$$\frac{C(n) = \text{sub } r_1 \ r_2 \ R(r_1) = n_1 \ R(r_2) = n_2 \ R' = R, r_1 \mapsto n_1 - n_2 \ n' = n + 1 \ F' = F, 0 \mapsto n_2 > n_1}{n; C; R; F; M; P \Rightarrow n'; C; R'; F'; M; P} \text{sub}$$

$$\frac{C(n) = \text{cmp } r_1 \ r_2 \ R(r_1) = n_1 \ R(r_2) = n_2 \ F' = F, 1 \mapsto n_1 == n_2 \ n' = n + 1}{n; C; R; F; M; P \Rightarrow n'; C; R; F'; M; P} \text{cmp}$$

$$\frac{C(n) = \text{jmp } r_1 \quad R(r_1) = n_1 \quad n' = n_1}{n; C; R; F; M; P \Rightarrow n'; C; R; F; M; P} \text{jmp}$$

$$\frac{C(n) = \text{jiz } r_1 \quad R(r_1) = n_1 \quad F(1) = b \quad n' = \text{if } b \text{ then } n_1 \text{ else } n + 1}{n; C; R; F; M; P \Rightarrow n'; C; R; F; M; P} \text{jiz}$$

$$\frac{C(n) = \text{jeq } r_1 \quad R(r_1) = n_1 \quad F(0) = b \quad n' = \text{if } b \text{ then } n_1 \text{ else } n + 1}{n; C; R; F; M; P \Rightarrow n'; C; R; F; M; P} \text{jeq}$$

$$\frac{C(n) = \text{read } r_1 \ r_2 \quad R(r_2) = n_2 \quad M(n_2) = n'' \quad n'' \geq 0 \quad \text{UNPROT}(P, n_2) \quad R' = R, r_1 \mapsto n'' \quad n' = n + 1}{n; C; R; F; M; P \Rightarrow n'; C; R'; F; M; P} \text{read}$$

$$\frac{C(n) = \text{write } r_1 \ r_2 \quad R(r_1) = n_1 \quad R(r_2) = n_2 \quad \text{UNPROT}(P, n_1) \quad M' = M, n_1 \mapsto n_2 \quad n' = n + 1}{n; C; R; F; M; P \Rightarrow n'; C; R; F; M'; P} \text{write}$$

$$\frac{C(n) = \text{cap } r \quad R(r) = n_1 \quad \text{UNPROT}(P, n_1) \quad i = \text{FREE}(P) \quad M' = M, n_1 \mapsto i \quad n' = n + 1}{n; C; R; F; M; P \Rightarrow n'; C; R; F; M'; P} \text{cap}$$

$$\frac{C(n) = \text{prot } r_1 \ r_2 \quad R(r_1) = n_1 \quad R(r_2) = n_2 \quad M(n_2) = i \quad i < 0 \quad \text{UNPROT}(P, n_1) \quad P' = P, i \mapsto n_1 \quad n' = n + 1}{n; C; R; F; M; P \Rightarrow n'; C; R; F; M; P'} \text{prot}$$

$$\frac{C(n) = \text{readp } r_1 \ r_2 \ r_3 \quad \begin{array}{l} R(r_2) = n_2 \\ R(r_3) = n_3 \end{array} \quad \begin{array}{l} M(n_2) = n'' \\ M(n_3) = i \end{array} \quad n'' \geq 0 \quad P(i) = n_2 \quad R' = R, r_1 \mapsto n'' \quad n' = n + 1}{n; C; R; F; M; P \Rightarrow n'; C; R'; F; M; P} \text{readp}$$

$$\frac{C(n) = \text{writep } r_1 \ r_2 \ r_3 \quad \begin{array}{l} R(r_1) = n_1 \\ R(r_2) = n_2 \\ R(r_3) = n_3 \end{array} \quad \begin{array}{l} M(n_3) = i \\ P(i) = n_1 \end{array} \quad M' = M, n_1 \mapsto n_2 \quad n' = n + 1}{n; C; R; F; M; P \Rightarrow n'; C; R; F; M', P} \text{writep}$$

The new commands are:

- *cap r*: generates a new capability and stores it in the memory address pointed by *r*
- *prot r₁ r₂*: protects the memory address stored in *r₁* with a capability stored in memory and pointed at by *r₂*
- *readp r₁ r₂ r₃*: reads the memory address stored in *r₂* and stores the content in *r₁*. Requires the capability pointed at by *r₃*
- *writep r₁ r₂ r₃*: writes the content of *r₂* in the memory address stored in *r₁*. Requires the capability pointed at by *r₃*

3 Typing CASM

Provide typing rules for all instructions of CASM (that is, all instructions from RML, ASM and CASM).

We can type each instruction by the arguments it receives and the type of side effect it has.

We τ to contain the side effects, which have no value

$\tau ::= \dots rw$	Write to register
$ sf$	Set flag
$ j$	Jump
$ mw$	Write to memory
$ cc$	Create capability
$ pm$	Protect memory with a capability

We also expand τ with the register type, and v with the registers defined for RML

$$\begin{aligned}\tau &::= \dots | \mathbb{R} \\ v &::= \dots | ar | br | \dots | or\end{aligned}$$

$$\begin{array}{lll}
\frac{\Gamma \vdash r : \mathbb{R} \quad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash r := n : \mathbb{R} \rightarrow \mathbb{N} \rightarrow rw} \text{load} &
\frac{\Gamma \vdash r_1 : \mathbb{R} \quad \Gamma \vdash r_2 : \mathbb{R}}{\Gamma \vdash sum\ r_1\ r_2 : \mathbb{R} \rightarrow \mathbb{R} \rightarrow rw} \text{sum} &
\frac{\Gamma \vdash r_1 : \mathbb{R} \quad \Gamma \vdash r_2 : \mathbb{R}}{\Gamma \vdash sub\ r_1\ r_2 : \mathbb{R} \rightarrow \mathbb{R} \rightarrow rw \times sf} \text{sub} \\
\frac{\Gamma \vdash r_1 : \mathbb{R} \quad \Gamma \vdash r_2 : \mathbb{R}}{\Gamma \vdash cmp\ r_1\ r_2 : \mathbb{R} \rightarrow \mathbb{R} \rightarrow sf} \text{cmp} &
\frac{\Gamma \vdash r : \mathbb{R}}{\Gamma \vdash jmp\ r : \mathbb{R} \rightarrow sf} \text{jmp} &
\frac{\Gamma \vdash r : \mathbb{R}}{\Gamma \vdash jiz\ r : \mathbb{R} \rightarrow sf} \text{jiz} \\
\frac{\Gamma \vdash r : \mathbb{R}}{\Gamma \vdash jeq\ r : \mathbb{R} \rightarrow sf} \text{jeq} &
\frac{\Gamma \vdash r_1 : \mathbb{R} \quad \Gamma \vdash r_2 : \mathbb{R}}{\Gamma \vdash read\ r_1\ r_2 : \mathbb{R} \rightarrow \mathbb{R} \rightarrow rw} \text{read} &
\frac{\Gamma \vdash r_1 : \mathbb{R} \quad \Gamma \vdash r_2 : \mathbb{R}}{\Gamma \vdash write\ r_1\ r_2 : \mathbb{R} \rightarrow \mathbb{R} \rightarrow mw} \text{write} \\
\frac{\Gamma \vdash r : \mathbb{R}}{\Gamma \vdash cap\ r : \mathbb{R} \rightarrow cc} \text{cap} &
\frac{\Gamma \vdash r_1 : \mathbb{R} \quad \Gamma \vdash r_2 : \mathbb{R}}{\Gamma \vdash prot\ r_1\ r_2 : \mathbb{R} \rightarrow \mathbb{R} \rightarrow pm} \text{prot} &
\frac{\Gamma \vdash r_1 : \mathbb{R} \quad \Gamma \vdash r_2 : \mathbb{R} \quad \Gamma \vdash r_3 : \mathbb{R}}{\Gamma \vdash prot\ r_1\ r_2\ r_3 : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \rightarrow pm} \text{readp} \\
&&
\frac{\Gamma \vdash r_1 : \mathbb{R} \quad \Gamma \vdash r_2 : \mathbb{R} \quad \Gamma \vdash r_3 : \mathbb{R}}{\Gamma \vdash prot\ r_1\ r_2\ r_3 : \mathbb{R} \rightarrow \mathbb{R} \rightarrow \mathbb{R} \rightarrow mw} \text{writep}
\end{array}$$

The types for each instruction work to represent their "shape" and behaviour.

4 Subtyping

Subtyping lets you use a term of type τ at a super-type τ' . Intuitively, it is the same principle by which you can use an object of a class as if it were of a super class.

Concretely, subtyping is achieved by introducing the subsumption rule below, as well as an ordering on types, indicated with $<:$.

$$\frac{\Gamma \vdash t : \tau \quad \tau <: \tau'}{\Gamma \vdash t : \tau'} \text{Subsumption}$$

Consider the standard typing of the sequencing rule and its related reduction. Suppose we extend our types with *Unit*, so $\tau ::= \dots | \text{Unit}$ and our terms with *unit* (a new value), so $t ::= \dots | \text{unit}$.

$$\frac{\Gamma \vdash t : \text{Unit} \quad \Gamma \vdash t' : \tau}{\Gamma \vdash t; t' : \tau} \text{Type-seq} \quad \frac{}{\text{unit}; t \rightsquigarrow^p t} \text{Eval-seq}$$

Consider STLC with sums and pairs. Can we introduce an ordering on types to allow this kind of reduction: $v; t \rightsquigarrow^p t$ for any value v , effectively replacing Rule **Eval-seq** while keeping Rule **Type-seq** and Rule **Subsumption**? If yes, show such an ordering. If no, argue why not.

$$\begin{aligned} \mathbb{N} &<: \text{Unit} \\ \tau \rightarrow \tau &<: \text{Unit} \\ \tau \uplus \tau &<: \text{Unit} \\ \tau \times \tau &<: \text{Unit} \end{aligned}$$

This ordering allows everything to be treated as *Unit*, thus we can type sequencing for any value on the left side.