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: $t ::= n|x|\lambda x.t|t\ t|\langle t,t\rangle|t.1|t.2|\ \mathit{inl}\ t|\ \mathit{inr}\ t|\text{case } t \text{ of } \left|\begin{array}{l}\mathit{inl}\ x_1 \to t\\ \mathit{inr}\ x_2 \to t\end{array}\right.$

Encoding these terms into lambdas is not an option.

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

Base cases

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

Inductive cases

$$F(t_1\ t_2) = \text{case unfold}(F(t_1)) \text{ of } \left|\begin{array}{l}\mathit{inl}\ x_1 \mapsto \text{case } x_1 \text{ of } \left|\begin{array}{l}\mathit{inl}\ x_{11} \mapsto \Omega\\ \mathit{inr}\ x_{12} \mapsto x_{12}\ F(t_2)\end{array}\right.\\ \\ \mathit{inr}\ x_2 \mapsto \Omega\end{array}\right.$$

$$F(t.1) = \text{case unfold}(F(t)) \text{ of } \left|\begin{array}{l}\mathit{inl}\ x_1 \mapsto \Omega\\ \\ \mathit{inr}\ x_2 \mapsto \text{case } x_2 \text{ of } \left|\begin{array}{l}\mathit{inl}\ x_{21} \mapsto \Omega\\ \mathit{inr}\ x_{22} \mapsto x_{22}.1\end{array}\right.\end{array}\right.$$

$$F(t.2) = \text{case unfold}(F(t)) \text{ of } \left|\begin{array}{l}\mathit{inl}\ x_1 \mapsto \Omega\\ \\ \mathit{inr}\ x_2 \mapsto \text{case } x_2 \text{ of } \left|\begin{array}{l}\mathit{inl}\ x_{21} \mapsto \Omega\\ \mathit{inr}\ x_{22} \mapsto x_{22}.2\end{array}\right.\end{array}\right.$$

$$F\left(\text{case } t_0 \text{ of } \left|\begin{array}{l}\mathit{inl}\ x_1 \mapsto t_1\\ \mathit{inr}\ x_2 \mapsto t_2\end{array}\right.\right) = \text{case unfold}(F(t_0)) \text{ of } \left|\begin{array}{l}\mathit{inl}\ x_1 \mapsto \Omega\\ \\ \mathit{inr}\ x_2 \mapsto \text{case } x_2 \text{ of } \left|\begin{array}{l}\mathit{inl}\ x_{2_1} \mapsto \text{case } x_{2_1} \text{ of } \left|\begin{array}{l}\mathit{inl}\ x_{211} \mapsto F(t_1)\\ \mathit{inr}\ x_{212} \mapsto F(t_2)\end{array}\right.\\ \\ \mathit{inr}\ x_{2_2} \mapsto \Omega\end{array}\right.\end{array}\right.$$

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.

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

Judgement:

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

Codebase, registers, flags, memory, and protections:

$$\frac{C = C', n \mapsto t}{C(n) = t} \qquad \frac{R = R', r \mapsto n}{R(r) = n} \qquad \frac{F = F', n \mapsto B}{F(n) = B}$$

$$\frac{C = C', n' \mapsto \_ \quad C'(n) = t}{C(n) = t} \qquad \frac{R = R', r' \mapsto \_ \quad R'(r) = n}{R(r) = n} \qquad \frac{F = F', n' \mapsto \_ \quad F'(n) = B}{F(n) = B}$$

$$\frac{M = M', n \mapsto n'}{M(n) = n'} \qquad \frac{M = M', n'' \mapsto \_ \quad M'(n) = n'}{M(n) = n'}$$

$$\frac{P = P', i \mapsto n}{P(i) = n} \qquad \frac{P = P', i' \mapsto \_ \quad P'(i) = n}{P(i) = n} \qquad \frac{P = P', i' \mapsto n' \quad P'(i) = n}{P(i) = n}$$

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

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

The UNPROT function checks if a memory address is unprotected

$$\text{UNPROT}(\emptyset, n) = true$$
$$\text{UNPROT}(P, i \to n, n) = false$$
$$\text{UNPROT}(P, i \to n', n) = UNPROT(P)$$

Rules:

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

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

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

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

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

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

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

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

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

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

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

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

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

The new commands are:

- *cap* r: generates a new capability and stores it in the memory address pointed by r
- *prot* r_1 r_2: protects the memory address stored in r_1 with a capability stored in memory and pointed at by r_2
- *readp* r_1 r_2 r_3: reads the memory address stored in r_2 and stores the content in r_1. Requires the capability pointed at by r_3
- *writep* r_1 r_2 r_3: writes the content of r_2 in the memory address stored in r_1. Requires the capability pointed at by r_3

3 Typing CASM

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

We can expand τ to include registers and side effects

$$\tau ::= \cdots |\mathcal{R}|sf|j|mw|cc|pm$$

The side effects are represented by types with no associated value

rw	Write to register
sf	Set flag
j	Jump
mw	Write to memory
cc	Create capability
pm	Protect memory with a capability

We define $\mathcal{S}$ as a generic side effect

$$\mathcal{S} \triangleq sf \uplus j \uplus mw \uplus cc \uplus pm$$

We define arguments as either registers or natural numbers

$$\mathcal{A} \triangleq \mathcal{R} \uplus \mathbb{N}$$

and a list of arguments as we defined lists in class

$$\mathcal{A}rgs \triangleq \mu\alpha.Bool \uplus (\mathcal{A} \times \alpha)$$

Finally each instruction can be typed as a pair containing a list of arguments and a side effect

$$\mathcal{I} \triangleq \mathcal{A}rgs \times \mathcal{S}$$

The values for $\mathcal{R}$ are the registers $ar, br, \ldots, or$ as defined for RML

The typing rules for the instructions are

$$\frac{\Gamma \vdash r : \mathcal{R} \quad \Gamma \vdash n : \mathbb{N}}{\Gamma \vdash r := n}\text{load}$$

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 $\tau ::= \cdots | Unit$ and our terms with $unit$ (a new value), so $t ::= \cdots | unit$.

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

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.

$$\begin{gathered} \mathbb{N} <: Unit \\ \tau \to \tau <: Unit \\ \tau \uplus \tau <: Unit \\ \tau \times \tau <: Unit \end{gathered}$$

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