
As before, you may work alone or in groups of two or three. This assignment is quite long and working in a group is highly recommended. If you work in a group, please form a group in CMS. Only one person needs to submit the assignment.

1. Dangling References

   In class we claimed that during evaluation, FL! programs never generate dangling references. Let's prove it. Consider the fragment of FL! consisting of the following expressions and values:

$$\begin{aligned} e \; &::= \; n \mid x \mid \mathsf{ref}\; e \mid !e \mid e_1 := e_2 \mid \mathsf{null} \mid \lambda x.\, e \mid e_0\; e_1 \\ &\quad\; \mid \mathsf{let}\; x = e_0 \;\mathsf{in}\; e_1 \mid (e_1, e_2) \mid \#1\, e \mid \#2\, e \\ v \; &::= \; n \mid (v_1, v_2) \mid \lambda x.\, e \text{ closed} \mid \mathsf{null} \end{aligned}$$

   To define the small-step semantics of FL! (Lecture 13), we augmented these with a set of *locations* $\ell \in Loc$.

$$e \; ::= \; \cdots \mid \ell \qquad\qquad v \; ::= \; \cdots \mid \ell$$

   A *store* σ is a partial map from locations to values (which could be other locations). The small-step semantics of FL! programs was defined in terms of *configurations* $\langle e,\, \sigma\rangle$, where e is an augmented expression and σ is a store. An FL! *program* is a closed expression not containing any ℓ.

   (a) Give an inductive definition of the set $\mathrm{loc}(e)$ of locations occurring in e.

   (b) Prove that if e is an FL! program and $\langle e,\, \varnothing\rangle \xrightarrow{*} \langle e',\, \sigma\rangle$, then $\mathrm{loc}(e') \subseteq \mathsf{dom}\,\sigma$. If you use induction, identify the relation you are using in your induction and argue that it is well-founded.

2. Projective Limits

   Let D, D' be CPOs. Recall from Lecture 24 that $D \precsim D'$ if there exists an *embedding-projection pair*, a pair of continuous functions $e : D \to D'$ and $p : D' \to D$ such that

$$p \circ e \;=\; \mathsf{id}_D \qquad\qquad e \circ p \;\sqsubseteq\; \mathsf{id}_{D'}. \tag{1}$$

   In the projective limit construction, we took a sequence of CPOs D_n such that $D_n \precsim D_{n+1}$ for all $n \geq 0$ with embedding-projection pairs $e_n : D_n \to D_{n+1}$ and $p_n : D_{n+1} \to D_n$ and formed the projective limit $\lim_n D_n$ consisting of all sequences $(d_n \mid n \geq 0) \in \Pi_n D_n$ such that $d_n = p_n(d_{n+1})$ for all $n \geq 0$. The ordering $\sqsubseteq$ on $\lim_n D_n$ is the ordering inherited from the product space $\Pi_n D_n$, namely the componentwise ordering.

   (a) Argue that $\lim_n D_n$ is a closed subspace of $\Pi_n D_n$ in the sense that the supremum of any chain in $\lim_n D_n$ is also in $\lim_n D_n$. Thus $\lim_n D_n$ is a CPO and is a sub-CPO of $\Pi_n D_n$.

   (b) Show that $D_m \precsim \lim_n D_n$ for all m by giving embedding-projection pairs $\widehat{e}_m : D_m \to \lim_n D_n$ and $\widehat{p}_m : \lim_n D_n \to D_m$ such that $\widehat{e}_m$ and $\widehat{p}_m$ are continuous and satisfy (1). (*Hint.* Your functions should satisfy the properties

$$\widehat{e}_{n+1} \circ e_n \;=\; \widehat{e}_n \qquad\qquad p_n \circ \widehat{p}_{n+1} \;=\; \widehat{p}_n.)$$

3. Exception Handling in Java

   In the next few exercises we will study the try-catch-finally construct of Java. Consider a language consisting of

- A set *Exc* of *exceptions* $E, F, \ldots$
- A set of *atomic commands* a
- A set of *tests* b
- A set *Com* of *commands* c given by the following grammar:

$$\begin{array}{rcl} c & ::= & a \mid \mathsf{skip} \mid \mathsf{throw}\ E \mid c_1\,;c_2 \mid \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2 \\ & & \mid \mathsf{try}\ c_1\ \mathsf{catch}\ E\ c_2\ \mathsf{finally}\ c_3 \mid \mathsf{try}\ c_1\ \mathsf{catch}\ E\ c_2 \mid \mathsf{try}\ c_1\ \mathsf{finally}\ c_3 \end{array}$$

Intuitively, try c_1 catch E c_2 finally c_3 works as follows. Statements may terminate normally or may terminate exceptionally by throwing an uncaught exception. The command c_2 is called an *exception handler* for E, and its scope is c_1. It is invoked if c_1 throws an uncaught exception E. If c_1 terminates normally, or if c_1 throws an exception $F \neq E$, then c_2 is never invoked.

The finally clause c_3 is always executed last, regardless of whether c_1 or c_2 terminate normally or exceptionally. Any uncaught exception other than E thrown by c_1 or any uncaught exception thrown by c_2 is rethrown upon normal termination of c_3. If c_3 throws an exception F, then c_3 terminates exceptionally and F is rethrown. In all other cases, the try command terminates normally.

Assume that with each atomic command a there is associated a total function $M_a : St \to St$ on some set of *states* St, and with each test b there is associated a total Boolean-valued function $M_b : St \to \mathbb{2}$. The exact nature of St, M_a, and M_b are unimportant.

We wish to define a big-step operational semantics for this language. Programs will be interpreted as partial functions $\mathcal{D} \rightharpoonup \mathcal{D}$, where $\mathcal{D} = St + (Exc \times St)$. We assume that St and $Exc \times St$ are disjoint and dispense with injection functions to simplify notation. Symbols $s, t, u, \ldots$ denote elements of St and $T, U, V, \ldots$ denote elements of $\mathcal{D}$. Intuitively, an output of the form (E, s) means that exception E has been thrown in state s; an output of the form s means that the program terminated normally in state s. The notation $\langle c,\, s\rangle \Downarrow T$ means that if command c is started in state s then it halts in output state T.

Here are the semantic rules for all constructs except the try command:

$$\langle \mathsf{skip},\, s\rangle \Downarrow s \qquad \langle a,\, s\rangle \Downarrow M_a(s) \qquad \langle c,\, (E,t)\rangle \Downarrow (E,t) \qquad \langle \mathsf{throw}\ E,\, s\rangle \Downarrow (E,s)$$

$$\frac{\langle c_1,\, s\rangle \Downarrow T \qquad \langle c_2,\, T\rangle \Downarrow U}{\langle c_1\,;c_2,\, s\rangle \Downarrow U} \qquad \frac{\langle c_1,\, s\rangle \Downarrow T \qquad M_b(s)}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2,\, s\rangle \Downarrow T} \qquad \frac{\langle c_2,\, s\rangle \Downarrow T \qquad \neg M_b(s)}{\langle \mathsf{if}\ b\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2,\, s\rangle \Downarrow T}$$

Here are the rules for the try command with either the catch clause or the finally clause omitted:

$$\frac{\langle c_1,\, s\rangle \Downarrow t}{\langle \mathsf{try}\ c_1\ \mathsf{catch}\ E\ c_2,\, s\rangle \Downarrow t} \qquad \frac{\langle c_1,\, s\rangle \Downarrow (E,t) \qquad \langle c_2,\, t\rangle \Downarrow U}{\langle \mathsf{try}\ c_1\ \mathsf{catch}\ E\ c_2,\, s\rangle \Downarrow U} \qquad \frac{\langle c_1,\, s\rangle \Downarrow (F,t),\ F \neq E}{\langle \mathsf{try}\ c_1\ \mathsf{catch}\ E\ c_2,\, s\rangle \Downarrow (F,t)}$$

$$\frac{\langle c_1,\, s\rangle \Downarrow t \qquad \langle c_3,\, t\rangle \Downarrow U}{\langle \mathsf{try}\ c_1\ \mathsf{finally}\ c_3,\, s\rangle \Downarrow U} \qquad \frac{\langle c_1,\, s\rangle \Downarrow (F,t) \qquad \langle c_3\,;\mathsf{throw}\ F,\, t\rangle \Downarrow U}{\langle \mathsf{try}\ c_1\ \mathsf{finally}\ c_3,\, s\rangle \Downarrow U}$$

Give rules for the full try-catch-finally command. There should be four.

4. Some Redundancy

   (a) The try-catch and try-finally constructs are redundant, since we can define them in terms of try-catch-finally. Conversely, given try-catch and try-finally, we can define try-catch-finally. Give these three definitions.

   (b) Java allows try commands with multiple handlers, e.g.

   $$\mathsf{try}\ c\ \mathsf{catch}\ E_1\ d_1\ \mathsf{catch}\ E_2\ d_2$$

   where $E_1 \neq E_2$. The scope of both handlers is c. Show that this construct is redundant by defining it in terms of the single-handler try command.

Note that the answer

$$\textsf{try try } c \textsf{ catch } E_1\ d_1 \textsf{ catch } E_2\ d_2$$

is incorrect, because an exception E_2 thrown by d_1 would be erroneously caught by d_2. (*Hint.* use a new exception not occurring in the command.)

(c) Show that finally is redundant in the presence of try-catch. That is, show how to define the construct try c catch E d finally e in terms of the construct try c catch E d. You may assume that your solution to the previous problem has been generalized to arbitrarily many handlers, and you may assume knowledge of all the exceptions that can be thrown by c.

5. Exceptions and Continuations

Now we will formulate a continuation-passing denotational semantics involving two continuations, a normal continuation $k : St \rightharpoonup St$ and an exceptional continuation $x : Exc \to St \rightharpoonup St$ to be invoked upon normal and exceptional termination, respectively. Define

$$NCont \triangleq St \rightharpoonup St \qquad XCont \triangleq Exc \to St \rightharpoonup St$$

The meaning function is

$$[\![\ ]\!] : Com \to NCont \to XCont \to St \rightharpoonup St.$$

The meanings of all commands except the try commands are defined as follows.

$$\begin{aligned}
[\![a]\!]\,k\,x &\triangleq k \circ M_a \\
[\![\textsf{skip}]\!]\,k\,x &\triangleq k \\
[\![\textsf{throw } E]\!]\,k\,x &\triangleq x\,E \\
[\![c_1\ ;\ c_2]\!]\,k\,x &\triangleq [\![c_1]\!]\,([\![c_2]\!]\,k\,x)\,x \\
[\![\textsf{if } b \textsf{ then } c_1 \textsf{ else } c_2]\!]\,k\,x &\triangleq \lambda s.\ \mathit{if}\ M_b(s)\ \mathit{then}\ [\![c_1]\!]\,k\,x\,s\ \mathit{else}\ [\![c_2]\!]\,k\,x\,s
\end{aligned}$$

The meanings of try-catch and try-finally can be defined as follows:

$$\begin{aligned}
[\![\textsf{try } c \textsf{ catch } E\ d]\!]\,k\,x &\triangleq [\![c]\!]\,k\,(\lambda F.\ \mathit{if}\ F = E\ \mathit{then}\ [\![d]\!]\,k\,x\ \mathit{else}\ x\,F) \\
[\![\textsf{try } c \textsf{ finally } e]\!]\,k\,x &\triangleq [\![c]\!]\,([\![e]\!]\,k\,x)\,(\lambda F.[\![e]\!]\,(x\,F)\,x)
\end{aligned}$$

Give a definition of the full try-catch-finally command.

6. Functions

We wish to add parameterless function calls to the language, along with dynamically-scoped exception handling. A *program* now consists of a finite sequence of function declarations

$$\textsf{function } f \textsf{ throws } E_1, \ldots, E_n\ \{c\},$$

at most one for each function name f, followed by a command. We also add a new atomic command call f. The declared functions may be mutually recursive.

$$\begin{aligned}
c &::= \ \ldots \mid \textsf{call } f \\
p &::= \ \textsf{function } f \textsf{ throws } E_1, \ldots, E_n\ \{c\}\ ;\ p \mid c
\end{aligned}$$

For example,

function f throws E {if b_1 then throw E else c_1 ; call g};
function g throws E {if b_2 then throw E else c_2 ; call f};
try call f catch E c_3

Any uncaught exception that could be thrown by the body of a function must be declared in the function declaration. If the body of a function throws an uncaught exception, then the function terminates exceptionally and the same exception is rethrown at the call site.

(a) Give the continuation-passing semantics of call f. Explain briefly what you would need to do to define the semantics when the declared functions are mutually recursive.

(b) Design a proof system for proving that all possible uncaught exceptions thrown by the body of a function are declared in the function declaration. Your proof system should have judgments of the form

$$\vdash c : \Delta \qquad \vdash f \text{ is ok}$$

where c is a command, Δ is a set of exceptions, and f is a declared function. Intuitively, $\vdash c : \Delta$ means that Δ contains all uncaught exceptions that can be thrown by c, and $\vdash f$ is ok means that the body of f does not throw any undeclared exceptions. One rule of your system will be

$$\frac{\vdash c : \Delta \qquad \Delta \subseteq \{E_1, \ldots, E_n\}}{\vdash f \text{ is ok}}$$

where the declaration of f is function f throws $E_1, \ldots, E_n$ $\{c\}$.

(c) Explain how the explicit declaration of thrown exceptions simplifies the previous problem.