Lecture notes on MiniML types and the Curry-Howard correspondence.

---

includes:

→ control operators

→ continuations

→ stack-based semantics

→ classical logic

References:

- Harper's _FoPLs_
- Griffin, "Formulae-as-Types Notion of Control" [POPL '90]
- Duba et al, "Typing 1st class continuations in ML" [POPL '91]
- Felleisen et al. "A syntactic theory of sequential control" [TCS '87]

J. Fix

CSCI 384

PoPL

Fall 2021

MiniML types

$\tau ::= \text{int} \mid \text{bool} \mid \text{unit}$

$\quad \mid \tau + \tau \mid \tau \times \tau \mid \tau \to \tau$

$e ::= 3 \mid \text{true} \mid ()$

$\quad \mid \text{fst } e \mid \text{snd } e \mid (e, e)$

$\quad \mid \text{lft } e \mid \text{rgt } e \mid \text{case } e \text{ of lft } x \Rightarrow e \mid \text{rgt } x \Rightarrow e$

$\quad \mid \text{let val } x = e \text{ in } e \text{ end}$

$\quad \mid \text{let fun } x\ x = e \text{ in } e \text{ end}$

$\quad \mid \text{fn } x \Rightarrow e \mid e\ e \mid x$

$\quad \mid e + e \mid e < e \mid e \text{ andalso } e$

MiniML type rules

$$\frac{}{\Gamma \vdash 3 : \text{int}} \qquad \frac{}{\Gamma \vdash \text{true} : \text{bool}} \qquad \frac{}{\Gamma \vdash () : \text{unit}}$$

$$\frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash \text{fst } e : \sigma} \qquad \frac{\Gamma \vdash e : \sigma \times \tau}{\Gamma \vdash \text{snd } e : \tau} \qquad \frac{\Gamma \vdash e_1 : \sigma \quad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1, e_2) : \sigma \times \tau}$$

$$\frac{\Gamma \vdash e : \sigma}{\Gamma \vdash \text{lft } e : \sigma + \tau} \qquad \frac{\Gamma \vdash e : \tau}{\Gamma \vdash \text{rgt } e : \sigma + \tau} \qquad \frac{\Gamma \vdash e : \sigma + \tau \quad \Gamma, x : \sigma \vdash e_1 : \rho \quad \Gamma, y : \tau \vdash e_2 : \rho}{\Gamma \vdash \text{case } e \text{ of lft } x \Rightarrow e_1 \mid \text{rgt } y \Rightarrow e_2 : \rho}$$

$$\frac{\Gamma \vdash d : \sigma \quad \Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \text{let val } x = d \text{ in } e \text{ end} : \tau} \qquad \frac{\Gamma, f : \sigma \to \tau, x : \sigma \vdash r : \tau \quad \Gamma, f : \sigma \to \tau \vdash e : \rho}{\Gamma \vdash \text{let fun } f\ x = r \text{ in } e \text{ end} : \rho}$$

$$\frac{\Gamma, x : \sigma \vdash e : \tau}{\Gamma \vdash \text{fn } x \Rightarrow e : \sigma \to \tau} \qquad \frac{\Gamma \vdash e_1 : \sigma \to \tau \quad \Gamma \vdash e_2 : \sigma}{\Gamma \vdash e_1\ e_2 : \tau} \qquad \frac{}{\Gamma \vdash x : \Gamma(x)}$$

$$\frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 + e_2 : \text{int}} \qquad \frac{\Gamma \vdash e_1 : \text{int} \quad \Gamma \vdash e_2 : \text{int}}{\Gamma \vdash e_1 < e_2 : \text{bool}} \qquad \frac{\Gamma \vdash e_1 : \text{bool} \quad \Gamma \vdash e_2 : \text{bool}}{\Gamma \vdash e_1 \text{ andalso } e_2 : \text{bool}}$$

Intuitionistic propositional logic (IL)

$$\frac{}{\Delta, A \vdash A} \qquad \frac{\Delta, A \vdash B}{\Delta \vdash A \Rightarrow B} \qquad \frac{\Delta \vdash A \quad \Delta \vdash A \Rightarrow B}{\Delta \vdash B}$$

$$\frac{\Delta \vdash A \wedge B}{\Delta \vdash A} \qquad \frac{\Delta \vdash A \wedge B}{\Delta \vdash B} \qquad \frac{\Delta \vdash A \quad \Delta \vdash B}{\Delta \vdash A \wedge B}$$

$$\frac{\Delta \vdash A}{\Delta \vdash A \vee B} \qquad \frac{\Delta \vdash B}{\Delta \vdash A \vee B} \qquad \frac{\Delta \vdash A \vee B \quad \Delta, A \vdash C \quad \Delta, B \vdash C}{\Delta \vdash C}$$

some theorems in IL

(*) $\quad (A \wedge B \Rightarrow C) \Rightarrow (A \Rightarrow (B \Rightarrow C))$

(‡) $\quad (A \Rightarrow (B \Rightarrow C)) \Rightarrow (A \wedge B \Rightarrow C)$

their proof terms

(*) $\vdash \text{fn } f \Rightarrow \text{fn } x \Rightarrow \text{fn } y \Rightarrow f\,(x, y) : ((\alpha \times \beta) \to \gamma) \to (\alpha \to (\beta \to \gamma))$

(‡) $\vdash \text{fn } f \Rightarrow \text{fn } p \Rightarrow (f\ (\text{fst } p))(\text{snd } p) : (\alpha \to (\beta \to \gamma)) \to (\alpha \times \beta \to \gamma)$

Curry-Howard

These MiniML terms' trees that establish their types correspond to analogous proofs in IL of their corresponding propositions (*) and (‡)

This showcases the Curry-Howard correspondence

context-based semantics

Let's give a stack-based semantics for evaluating MiniML terms. We first define values:

$$v ::= num(n) \mid cnd(b) \mid fn(x,e)$$

Then we define evaluation contexts:

$$\begin{aligned} C ::= {} & plus(\circ,e) \mid plus(v,\circ) \\ & \mid apply(\circ,e) \mid apply(v,\circ) \\ & \mid let(x,\circ,e) \mid if(\circ,e,e) \end{aligned}$$

Finally, we define stacks:

$$\Sigma ::= \bullet \mid \Sigma ; C$$

With these we define a relation

$$S_1 \twoheadrightarrow S_2 \quad \text{where} \quad S ::= \Sigma \rhd e \mid \Sigma \lhd v$$

by the rules:

$$\Sigma \rhd plus(e_1,e_2) \twoheadrightarrow \Sigma; plus(\circ,e_2) \rhd e_1$$

$$\Sigma ; plus(\circ,e) \lhd v \twoheadrightarrow \Sigma; plus(v,\circ) \rhd e$$

$$\Sigma ; plus(num(n_1),\circ) \lhd num(n_2) \twoheadrightarrow \Sigma \lhd num(n_1+n_2)$$

$$\Sigma \rhd if(e,e_1,e_2) \twoheadrightarrow \Sigma; if(\circ,e_1,e_2) \rhd e$$

$$\Sigma; if(\circ,e_1,e_2) \lhd cnd(true) \twoheadrightarrow \Sigma \rhd e_1$$

$$\Sigma; if(\circ,e_1,e_2) \lhd cnd(false) \twoheadrightarrow \Sigma \rhd e_2$$

$$\Sigma \rhd let(x,d,e) \twoheadrightarrow \Sigma; let(x,\circ,e) \rhd d$$

$$\Sigma; let(x,\circ,e) \lhd v \twoheadrightarrow \Sigma \rhd [v/x]e$$

$$\Sigma \rhd apply(e_1,e_2) \twoheadrightarrow \Sigma; apply(\circ,e_2) \rhd e_1$$

$$\Sigma; apply(\circ,e) \lhd v \twoheadrightarrow \Sigma; apply(v,\circ) \rhd e$$

$$\Sigma; apply(fn(x,e),\circ) \lhd v \twoheadrightarrow \Sigma \rhd [v/x]e$$

$$\Sigma \rhd v \twoheadrightarrow \Sigma \lhd v$$

control operators

Let's introduce a few more constructs to MiniML

$e ::= \text{catch } \kappa \text{ in } e \mid \text{throw } e_1 \text{ at } e_2 \mid \text{abort } e$

The "catch" term introduces an "escape hatch" through which evaluation can be stopped within e, "throwing" a value back out to the context where the catch resides. It is similar to an exception, and is a kind of "jump" for FPLs. The name k is often called a continuation. The "throw" term performs that "escape" by feeding a value to some continuation. "Catch" is "letcc" in the literature.

Here are the semantics:

- We extend values with

$v ::= \text{resume}(\Sigma)$

- We extend contexts with

$C ::= \text{throw}(e, \circ) \mid \text{throw}(\circ, v) \mid \text{abort}(\circ)$

- We extend the rules for $\twoheadrightarrow$ with

$$\Sigma \triangleright \text{catch}(\kappa, e) \twoheadrightarrow \Sigma \triangleright [\text{resume}(\Sigma)/\kappa]\, e$$

$$\Sigma \triangleright \text{throw}(e_1, e_2) \twoheadrightarrow \Sigma; \text{throw}(e_1, \circ) \triangleright e_2$$

$$\Sigma; \text{throw}(e, \circ) \triangleleft v \twoheadrightarrow \Sigma; \text{throw}(\circ, v) \triangleright e$$

$$\Sigma; \text{throw}(\circ, \text{resume}(\hat{\Sigma})) \triangleleft v \twoheadrightarrow \hat{\Sigma} \triangleleft v$$

$$\Sigma \triangleright \text{abort}(e) \twoheadrightarrow \Sigma; \text{abort}(\circ) \triangleright e$$

$$\Sigma; \text{abort}(\circ) \triangleleft v \twoheadrightarrow \bullet \triangleleft v$$

We see that "catch" captures the context Σ, and "throw" resumes some context $\hat{\Sigma}$ with the tossed value.

Example use of control operators

The function below computes the product of a list of integers, but exits out early with a "throw" if a 0 is hit.

```
fun productHelper k l =
    if null l then 1
    else if (hd l)=0 then (throw 0 at k)
                     else (hd l) * (productHelper k (tl l))

fun product l =
    catch k in (productHelper k l)
```

Here is a similar function, but it is tail recursive and also it escapes in all cases, i.e. all evaluation "paths".

```
fun productHelper k l p
    if null l then (throw p at k)
    else if (hd l)=0 then (throw 0 at k)
                     else (productHelper k (tl l) ((hd l)*p))

fun product l =
    catch k in (productHelper k l 1)
```

A brief note on their addition

These operations, and similar ones, were introduced to provide features to FPLs that might possibly enable others, for example, things like
- multi-threading
- co-routines
- exceptions
- callbacks

They also provide, in a first class way a mechanism that is mimicked by "continuation passing style" (or CPS), one that was being adopted by some FPL programmers for efficient coding, and for FPL compilation.

We'll consider CPS later, and first play a bit with control.

Another
Example use of control operators

This code outputs a string that normally gives the division of 100 by some integer input. But there are two exceptional cases, and these are handled by "throw" operations.

- a division by 0
- entry of a string that is not an integer

```
let
  fun digitOf err c = if c<'0' orelse c>'9'
                      then (throw "not an int" at err)
                      else ord(c)-ord('0')
  fun getInteger err m =
    let val c = getChar()
    in if c='\n' then m
       else let val d = digitOf err c
            in (getInteger err((m*10)+d))
            end
    end
  fun divideByInput err n =
    let val m = getInteger err 0
    in
      if m=0 then (throw "division by zero" at err)
      else intToString (n div m)
    end
in
  catch err in (print (divideByInput err 100))
end
```

Example (cont'd)

The stack-based semantics behaves differently in evaluating the sample expression in three scenarios:

A entering an integer that is not ∅

B entering 0.

C entering a character that is not a digit

In each scenario, the state looks like this just before the top-level call to divideByInput:

$\bullet; \text{print}(\circ) \triangleright \text{apply}(\text{apply}(\text{dBI}, \text{resume}(\bullet; \text{print}(\circ))), \text{num}(100))$

Here, dBI is the term for the divideByInput function.

Notice that err will be the continuation

$$\text{resume}(\bullet; \text{print}(\circ))$$

This is the "captured context" awaiting a string to be thrown to it

- Let's consider each of the scenarios, in turn:

A They enter the integer 42. In that case the variable m will bound to 42 like so

$$\bullet; \text{print}(\circ); \text{let}(m, \circ, \text{if}(\ldots)) \triangleleft \text{num}(42)$$

and evaluation continues as

$\rightarrow\!\!\!\!\rightarrow \bullet; \text{print}(\circ) \triangleright \text{if}(\text{equals}(\text{num}(42), \text{num}(\emptyset)), \ldots, \ldots)$

... (several steps omitted)

$\rightarrow\!\!\!\!\rightarrow \bullet; \text{print}(\circ); \text{apply}(\text{iTS}, \circ) \triangleright \text{div}(\text{num}(100), \text{num}(42))$

...

$\rightarrow\!\!\!\!\rightarrow \bullet; \text{print}(\circ) \triangleleft \text{txt}(\text{"2"})$

and so we output the result string of the division: "2"

(B) Suppose instead they enter 0
Then the stack state is similar

• ; print(0); let(m, 0, if(...)) ◁ num(0)

But instead the equals test will return cnd(true) and this will lead to the throw

...→> • ; print(0); if(0, ..., ...) ◁ cnd(true)

→> • ; print(0) ▷ throw(txt("division by zero"), resume(• ; print(0)))

→> • ; print(0) ; throw(txt("division by zero"), •) ▷ resume(• ; print(0))

→> • ; print(0) ; throw(txt("division by zero"), •) ◁ resume(• ; print(0))

→> • ; print(0) ; throw(•, resume(• ; print(0))) ▷ txt("division by zero")

→> • ; print(0) ; throw(•, resume(• ; print(0))) ◁ txt("division by zero")

Now that we've resolved the value to be thrown, we throw out the underlined context and resume execution with the captured context that's the target of the "throw." This leads to

→> • ; print(0) ◁ txt("division by zero")

which outputs the error message.

(C) Suppose instead they enter "a" instead of "42" or "0"
Then, within digitOf we obtain this state

• ; print(0); let(m, 0, if(...)); let(d, 0, let(m', ...)) ▷ throw(txt("not an int"), resume(• ; print(0)))

So, just like in the last example, the mechanism takes a few steps to figure out that both subterms of throw are already simplified to values. That leads to the state

...→> • ; print(0) ◁ txt("not an int")

This replaces the stacked up context underlined above.

control operator typing rules

- We add a new type to the system

  $\tau ::= \sim\tau$

  This type is $cont(\tau)$. It is the type of a continuation awaiting a value of type τ.

- We add these typing rules

$$\frac{\Gamma, k:\sim\tau \vdash e : \tau}{\Gamma \vdash \text{catch } k \text{ in } e : \tau} \qquad \frac{\Gamma \vdash e_2 : \sim\tau \quad \Gamma \vdash e_1 : \tau}{\Gamma \vdash \text{throw } e_1 \text{ at } e_2 : \sigma}$$

- Note that, since a throw leads to an "escape", it can be placed in any context. It can have any type σ. Furthermore, the expectation is that the value thrown has a type τ that the $\sim\tau$ continuation awaits.
- Note also that the type of the "catch" expression e matches the type τ expected to be thrown to the continuation k. It turns out that not all "paths" to possible resulting values of e have to escape. This is because e can just evaluate normally to a value of type τ.

alternatives

Historically, control was provided by a construct callcc
The use callcc(fn $k \Rightarrow e$) is equivalent to catch k in e
There is an alternative control(fn $k \Rightarrow e$) in the literature. For it, e never returns. All paths in e throw/escape
See Felleisen et al. [Theoretical CS '87] for some details.

Curry-Howard w/ control

- We introduce another type void, one that has no values:

$$\tau ::= \bot$$

- We change the type discipline of throw so that it yields type void (i.e. never returns).

$$\frac{\Gamma \vdash e_2 : \sim\tau \qquad \Gamma \vdash e_1 : \tau}{\Gamma \vdash \text{throw } e_1 \text{ at } e_2 : \bot}$$

- Furthermore, we insist that all evaluation paths within a catch expression lead to an escape/throw.

$$\frac{\Gamma, k : \sim\tau \vdash e : \bot}{\Gamma \vdash \text{catch } k \text{ in } e : \tau}$$

- And then, for a certain completeness, we include

$$\frac{\Gamma \vdash e : \bot}{\Gamma \vdash \text{abort}(e) : \sigma}$$

Squinting at these rules, we get a correspondence with

Classical Logic (CL)

$$\frac{\Delta, \neg A \vdash \bot}{\Delta \vdash A} \qquad \frac{\Delta \vdash \neg A \quad \Delta \vdash A}{\Delta \vdash \bot} \qquad \frac{\Delta \vdash \bot}{\Delta \vdash A}$$

These allow additional proofs, including proof by contradiction, and we give meaning to logical negation and absurdity

Some theorems of CL

$(A \Rightarrow \bot) \Rightarrow \neg A$

$\neg A \Rightarrow (A \Rightarrow \bot)$

$(A \Rightarrow B) \Rightarrow (\neg B \Rightarrow \neg A)$

$(\neg B \Rightarrow \neg A) \Rightarrow (A \Rightarrow B)$

$\neg\neg A \Rightarrow A$] double negation elimination (DNE)

$A \vee \neg A$] law of the excluded middle (LEM)

The proof terms (using MiniML + control):

$\vdash \mathrm{fn}\ k \Rightarrow \mathrm{catch}\ \varepsilon\ \mathrm{in}\ k(\mathrm{catch}\ \kappa\ \mathrm{in}\ (\mathrm{throw}\ \kappa\ \mathrm{at}\ \varepsilon)) : (\alpha \to \bot) \to \sim\alpha$

$\vdash \mathrm{fn}\ \kappa \Rightarrow \mathrm{fn}\ x \Rightarrow \mathrm{throw}\ x\ \mathrm{at}\ \kappa : \sim\alpha \to (\alpha \to \bot)$

$\vdash \mathrm{fn}\ f \Rightarrow \mathrm{fn}\ \ell \Rightarrow \mathrm{catch}\ \varepsilon\ \mathrm{in}\ (\mathrm{throw}\ f(\mathrm{catch}\ \kappa\ \mathrm{in}\ \mathrm{throw}\ \kappa\ \mathrm{at}\ \varepsilon)\ \mathrm{at}\ \ell) : (\alpha \to \beta) \to (\sim\beta \to \sim\alpha)$

$\vdash \mathrm{fn}\ T \Rightarrow \mathrm{fn}\ x \Rightarrow \mathrm{catch}\ \ell\ \mathrm{in}\ \mathrm{throw}\ x\ \mathrm{at}\ T\ell : (\sim\beta \to \sim\alpha) \to (\alpha \to \beta)$

$\vdash \mathrm{fn}\ \varepsilon \Rightarrow \mathrm{catch}\ \kappa\ \mathrm{of}\ \mathrm{throw}\ \kappa\ \mathrm{at}\ \varepsilon : \sim\sim\alpha \to \alpha$

$\vdash \mathrm{catch}\ \theta\ \mathrm{in}\ \mathrm{throw}\ \mathrm{lft}(\mathrm{catch}\ \kappa\ \mathrm{in}\ (\mathrm{throw}\ (\mathrm{rgt}\ \kappa)\ \mathrm{at}\ \theta))\ \mathrm{at}\ \theta : \alpha + \sim\alpha$

Here, for example, is the proof tree for the last:

$$
\dfrac{\dfrac{}{\neg(A\vee\neg A) \vdash \neg(A\vee\neg A)} \qquad \dfrac{\dfrac{\dfrac{\dfrac{}{\neg(A\vee\neg A), \neg A \vdash \neg(A\vee\neg A)} \qquad \dfrac{\dfrac{}{\neg(A\vee\neg A), \neg A \vdash \neg A}}{\neg(A\vee\neg A), \neg A \vdash A \vee \neg A}}{\neg(A\vee\neg A), \neg A \vdash \bot}}{\neg(A\vee\neg A) \vdash A}}{\neg(A\vee\neg A) \vdash A\vee\neg A}}{\dfrac{\neg(A\vee\neg A) \vdash \bot}{\vdash A \vee \neg A}}
$$