Sequential Programs

So far, the language that we’ve implemented is deterministic.

• Running a program multiple times (or computing things slightly more quickly or slowly) does not change the result of the program.

• Real programming languages do not behave this way
Threads

\[
\text{\{seqn \{spawn \textit{EXPR}_1\}} \\
\text{\{spawn \textit{EXPR}_2\}\}}
\]

Runs \textit{EXPR}_1 and \textit{EXPR}_2 in any order, even interleaved with each other
Threads

```plaintext
{with {b {struct {x 1}}}}
  {seqn {spawn {set b x 2}}
    {seqn {spawn {set b x 3}}
      {get b x}}}}
```

What are the possible results for the last expression?
Threads

```plaintext
{with {b {struct {x 1}}}
  {seqn {spawn {set b x 2}}
    {seqn {spawn {set b x 3}}
      {get b x}}}}
```

What are the possible results for the last expression?

1, 2, or 3
Threads

{with {b {struct {x 1}}}}
  {seqn {spawn {set b x 2}}
    {seqn {spawn {set b x 3}}
      {get b x}}}}

What are the possible results for the last expression?

1, 2, or 3

What about the other threads?
Threads

```{with {b {struct {x 1}}}}
  {seqn {spawn {set b x 2}}
    {seqn {spawn {set b x 3}}
      {get b x}}}}}
```

What are the possible results for the last expression?

1, 2, or 3

What about the other threads?

3 or 1 for (textually) first thread

2 or 1 for (textually) second thread
TRFAE = FAE + (Mutable) Records + Threads

<TRFAE> ::= <num>
    | [+ <TRFAE> <TRFAE>]
    | [- <TRFAE> <TRFAE>]
    | <id>
    | {fun {<id>} <TRFAE>}
    | {<TRFAE> <TRFAE>}
    | {struct {<id> <TRFAE>} ...}
    | {set <TRFAE> <id> <TRFAE>}
    | {get <TRFAE> <id>}
    | {spawn <TRFAE>}
    | {receive}
    | {deliver <TRFAE> <TRFAE>}
    | {seqn <TRFAE> <TRFAE>}
Block the current thread until a value is delivered to it:

\textbf{receive}

Send the value of \texttt{DELIVERABLE-EXPR} to \texttt{THD-EXPR} (which is expected to be a thread):

\texttt{deliver THD-EXPR DELIVERABLE-EXPR}

Does \textit{not} wait for receipt.
{with {t {spawn [+ 3 {receive}]}}}
{deliver t 2}
{with {t {spawn [+ 3 {receive}]}}}
{deliver t 2}

Spawned thread produces 5, the other produces 2
{with {t {spawn 1}}
 {prog {deliver t 2}
  {0 0}}}
{with {t {spawn 1}}
  {prog {deliver t 2}
    {0 0}}}

Raises an error even though the value is never delivered
{seqn {spawn {{fun {x} {x x}}
              {fun {x} {x x}}} }}
{0 0}}

{seqn {spawn {0 0}}
  {{fun {x} {x x}}
   {fun {x} {x x}}}}
Both programs must raise an error

Critical question: how can we interrupt our interpreter?
Continuation-passing style

**Key idea:** convert the interpreter into a style where all remaining work is explicit as an argument to the interpreter

Then we can swap in and out different pieces of work to swap between different threads
Continuation-passing style

Transform interpreter from:

\[
\text{interp} : (\rightarrow \text{TRFAE DefrdSub TRFAE-Value})
\]

into a function with this type:

\[
(\rightarrow \text{TRFAE DefrdSub} \ (\text{TRFAE-Value} \rightarrow \alpha) \ \alpha)
\]
Continuation-passing style

Transform interpreter from:

\[
\text{interp} : (\rightarrow \text{TRFAE DefrdSub TRFAE-Value})
\]

into a function with this type:

\[
(\rightarrow \text{TRFAE DefrdSub (TRFAE-Value} \rightarrow \alpha) \alpha)
\]

NB: the store is a result: so where does it go?
Continuation-passing style

Transform interpreter from:

\[
\text{interp} : (\rightarrow \text{TRFAE DefrdSub TRFAE-Value})
\]

into a function with this type:

\[
(\rightarrow \text{TRFAE DefrdSub} (\text{TRFAE-Value} \rightarrow \alpha) \alpha)
\]

NB: the store is a result: so where does it go?

\[
\text{interp} : (\rightarrow \text{TRFAE DefrdSub DefrdSub Store (TRFAE-Value*Store} \rightarrow \alpha) \alpha)
\]
What follows in the FAE interpreter, transformed in continuation-passing style. Each future step of computation is explicitly packaged up into a more complex $k$ argument to be supplied to the next call to interp
(define-type FAE
  [num (n number?)])
[add (lhs FAE?)
  (rhs FAE?)]
[sub (lhs FAE?)
  (rhs FAE?)]
[id (name symbol?)]
[fun (param symbol?)
  (body FAE?)]
[app (fun-expr FAE?)
  (arg-expr FAE?)])
(define-type FAE-Value
  [numV (n number?)]
  [cloV (param symbol?)
    (body FAE?)
    (ds DefrdSub?)])

(define-type DefrdSub
  [mtSub]
  [aSub (name symbol?)
    (value FAE-Value?)
    (rest DefrdSub?)])
(define (interp-expr a-fae)
  (type-case FAE-Value (interp a-fae
                       (mtSub)
                       (λ (x) x))
              [numV (n) n]
              [cloV (p b d) 'fun]))
(define/contract (interp a-fae ds k)
  (-> FAE? DefrdSub? (-> FAE-Value? any) any)
(type-case FAE a-fae
  [num (n) (k (numV n))]
  [add (l r) (numop + l r ds k)]
  [sub (l r) (numop - l r ds k)]
  [id (name) (k (lookup name ds))]
  [fun (param body-expr)
     (k (cloV param body-expr ds))]
  [app (fun-expr arg-expr)
     the next slide contains this case]))
...
(define (numop f l r ds k)
  (interp l ds
    (λ (l-v)
      (interp r ds
        (λ (r-v)
          (k (numV
            (f (numV-n l-v)
              (numV-n r-v))))))))))
(define (lookup name ds)
  (type-case DefrdSub ds
    [mtSub () (error 'lookup "free variable")]
    [aSub (sub-name num rest-ds)
      (if (symbol=? sub-name name)
        num
        (lookup name rest-ds))]]))