Type Systems For: Exceptions, Continuations, and Recursive Types

Exceptions
- A mechanism that allows non-local control flow
  - Useful for implementing the propagation of errors to caller
- Exceptions ensure* that errors are not ignored
  - Compare with the manual error handling in C
- Languages with exceptions:
  - C++, ML, Modula-3, Java, C#, ...
- We assume that there is a special type \texttt{exn} of exceptions
  - \texttt{exn} could be int to model error codes
  - In Java or C++, \texttt{exn} is a special object type

Modeling Exceptions
- Syntax
  \[ e ::= ... | \text{raise } e | \text{try } e_1 \text{ handle } x \Rightarrow e_2 \]
  \[ \tau ::= ... | \text{exn} \]
- We ignore here how exception values are created
  - In examples we will use integers as exception values
- The handler binds \( x \) in \( e_2 \) to the actual exception value
- The “\text{raise}” expression never returns to the immediately enclosing context
  - \( 1 + \text{raise } 2 \) is well-typed
  - if (\text{raise } 2) then 1 else 2 is also well-typed
  - (\text{raise } 2) 5 is also well-typed
  - What should be the type of \text{raise}?  

Example with Exceptions
- A (strange) factorial function
  \[
  \text{let } f = \lambda x: \text{int.} \lambda res: \text{int.} \text{if } x = 0 \text{ then raise res else} f (x - 1) (res \times x) \in \text{try } f 5 1 \text{ handle } x \Rightarrow x
  \]
- The function returns in one step from the recursion
- The top-level handler catches the exception and turns it into a regular result

Typing Exceptions
- New typing rules
  \[
  \Gamma \vdash e : \text{exn} \quad \Gamma \vdash \text{raise } e : \tau
  \]
  \[
  \Gamma \vdash e_1 : \tau \quad \Gamma, x: \text{exn} \vdash e_2 : \tau
  \]
  \[
  \Gamma \vdash \text{try } e_1 \text{ handle } x \Rightarrow e_2 : \tau
  \]
- A raise expression has an arbitrary type
  - This is a clear sign that the expression does not return to its evaluation context
  - The type of the body of try and of the handler must match
  - Just like for conditionals

Dynamics of Exceptions
- The result of evaluation can be an uncaught exception
  - Evaluation answers:
    \[ a ::= v | \text{uncaught } v \]
  - “uncaught \( v \)” has an arbitrary type
- Raising an exception has global effects
- It is convenient to use contextual semantics
  - Exceptions propagate through some contexts but not through others
  - We distinguish the handling contexts that intercept exceptions (this will be new)
Contexts for Exceptions

- **Contexts**
  - $H ::= \vdash | H e | v H | \text{raise } H | \text{try } H \ \text{handle } x \Rightarrow e$

- **Propagating contexts**
  - Contexts that propagate exceptions to their own enclosing contexts
  - $P ::= \vdash | P e | v P | \text{raise } P$

- **Decomposition theorem**
  - If $e$ is not a value and $e$ is well-typed then it can be decomposed in exactly one of the following ways:
    - $H[\lambda x: \tau. e] v$ (normal lambda calculus)
    - $H[\text{try } v \ \text{handle } x \Rightarrow e]$ (handle it or not)
    - $H[\text{try } P[\text{raise } v] \ \text{handle } x \Rightarrow e]$ (propagate!)
    - $P[\text{raise } v]$ (uncaught exception)

Contextual Semantics for Exceptions

- **Small-step reduction rules**
  - $H[\lambda x: \tau. e] v$ $\rightarrow$ $H[v/x] e$
  - $H[\text{try } v \ \text{handle } x \Rightarrow e]$ $\rightarrow$ $H[v]$
  - $H[\text{try } P[\text{raise } v] \ \text{handle } x \Rightarrow e]$ $\rightarrow$ $H[v/x] e$
  - $P[\text{raise } v]$ $\rightarrow$ uncaught $v$

- The handler is ignored if the body of try completes normally
- A raised exception propagates (in one step) to the closest enclosing handler or to the top of the program

Exceptional Commentary

- The addition of exceptions preserves type soundness
- Exceptions are like *non-local goto*
- However, they cannot be used to implement recursion
  - Thus we still cannot write (well-typed) non-terminating programs
- There are a number of ways to implement exceptions (e.g., “zero-cost” exceptions)

Some languages have a mechanism for taking a snapshot of the execution and storing it for later use
- Later the execution can be reinstated from the snapshot
- Useful for implementing threads, for example
- Examples: Scheme, LISP, ML, C (yes, really!)
- Consider the expression: $e_1 + e_2$ in a context $C$
  - How to express a snapshot of the execution right after evaluating $e_1$
  - But before evaluating $e_2$ and the rest of $C$?
    - Idea: as a context $C_{e_1} = C[\_ + e_2]$
      - Alternatively, as $\lambda x. C[x + e_2]$
    - When we finish evaluating $e_1$ to $v_1$, we fill the context and continue with $C[v_1 + e_2]$
      - But the $C_{e_1}$ continuation is still available and we can continue several times, with different replacements for $e_1$

Continuation Uses in “Real Life”

- You’re walking and come to a fork in the road
- You save a continuation “right” for going right
- But you go left (with the “right” continuation in hand)
- You encounter Bender. Bender coerces you into joining his computer dating service.
- You save a continuation “bad-date” for going on the date.
- You decide to invoke the “right” continuation
- So, you go right (no evil date obligation, but with the “bad-date” continuation in hand)
- A train hits you!
- On your last breath, you invoke the “bad-date” continuation

Continuations

- **Syntax**
  - $e ::= \text{callcc } k \ \text{in } e | \text{throw } e_1 e_2$
  - $\tau ::= \_ | \_ \tau$

- $\tau$ cont - the type of a continuation that expects a $\tau$
- callcc $k$ in $e$ - sets $k$ to the current context of the execution and then evaluates expression $e$
  - when $e$ terminates, the whole callcc terminates
  - $e$ can invoke the saved continuation (many times even)
  - when $e$ invokes $k$ it is as if “callcc $k$ in $e$” returns
    - $k$ is bound in $e$
  - throw $e_1 e_2$ - evaluates $e_1$ to a continuation, $e_2$ to a value and invokes the continuation with the value of $e_2$
    (just wait, we’ll explain it!)
Example with Continuations

- Example: another strange factorial

  callcc k in
  let f = \x:int.\res:int. if x = 0 then throw k res
  else f (x - 1) (x * res)
  in f 5 1

  • First we save the current context
  - This is the top-level context
  - A throw to k of value v means “pretend the whole callcc evaluates to v”
  • This simulates exceptions
  • Continuations are strictly more powerful than exceptions
    - The destination is not tied to the call stack

Static Semantics of Continuations

\[ \Gamma, k : \tau \text{ cont } \vdash e : \tau \]
\[ \Gamma \vdash \text{callcc } k \text{ in } e : \tau \]
\[ \Gamma \vdash e_1 : \tau \text{ cont } \Gamma \vdash e_2 : \tau \]
\[ \Gamma \vdash \text{throw } e_1 e_2 : \tau' \]

• Note that the result of callcc is of type \( \tau \)
  “callcc k in e” returns in two possible situations
  1. e throws to k a value of type \( \tau \), or
  2. e terminates normally with a value of type \( \tau \)
• Note that throw has any type \( \tau' \)
  - Since it never returns to its enclosing context

Dynamic Semantics of Continuations

• Use contextual semantics (wow, again!)
  - Contexts are now manipulated directly
  - Contexts are values of type \( \tau \text{ cont} \)
• Contexts
  \[ H ::= \varepsilon | H e | v H | \text{throw } H_1 e_2 | \text{throw } v, H_2 \]
• Evaluation rules
  - \( H[(\lambda x.e) v] \rightarrow H[\lambda x.e] \)
  - \( H[\text{callcc } k \text{ in } e] \rightarrow H[\text{callcc } k \text{ in } e] \)
  - \( H[\text{throw } H_1 v_2] \rightarrow H[v_2] \)
  • callcc duplicates the current continuation
  • Note that throw abandons its own context

Implementing Coroutines with Continuations

• Example:

  let client = \k. let res = callcc k' in throw k k' in
  print (fst res); client (snd res)

  • “client k” will invoke “k” to get an integer and a continuation for
  obtaining more integers
  (for now, assume the list & recursion work)

  let getnext = \L. \k. if L = nil then raise 999
  else getnext (cdr L) (callcc k' in throw k (car L, k'))

  • “getnext L k” will send to “k” the first element of L along with a
  continuation that can be used to get more elements of L.

  getnext [0;1;2;3;4;5] (callcc k in client k)

Continuation Comments

• In our semantics the continuation saves the entire context: program counter, local variables, call
  stack, and the heap!
• In actual implementations the heap is not saved!
• Saving the stack is done with various tricks, but it is expensive in general
• Few languages implement continuations
  - Because their presence complicates the whole compiler considerably
  - Unless you use a continuation-passing-style of
    compilation (more on this next)

Continuation Passing Style

• A style of compilation where evaluation of a function never returns directly: instead the
  function is given a continuation to invoke with its result.
• Instead of
  \[ \text{f(int a) \{} \text{return h(g(e));} \}\]
  we write
  \[ \text{f(int a, cont k) \{} g(e, \lambda r. h(r, k))} \]\n• Advantages:
  - interesting compilation scheme (supports callcc easily)
  - no need for a stack, can have multiple return addresses
    (e.g., for an error case)
  - fast and safe (non-preemptive) multithreading
Continuation Passing Style

- Let $e ::= x \mid n \mid e_1 + e_2 \mid \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \mid \lambda x.e \mid e_1 e_2$
- Define $cps(e, k)$ as the code that computes $e$ in CPS and passes the result to continuation $k$
  - $cps(x, k) = k x$
  - $cps(n, k) = k n$
  - $cps(e_1 + e_2, k) = cps(e_1, \lambda n_1. cps(e_2, \lambda n_2. k (n_1 + n_2)))$
  - $cps(\lambda x.e, k) = k (\lambda x. \lambda k'.cps(e, k'))$
  - $cps(e_1 e_2, k) = cps(e_1, \lambda f_1. cps(e_2, \lambda v_2. f_1 v_2 k))$
- Example: $cps(h(g(5)), k) = g(5, \lambda x. h x k)$
  - Notice the order of evaluation being explicit

Recursive Types: Lists

- We want to define recursive data structures
- Example: lists
  - A list of elements of type $\tau$ (a $\tau$ list) is either empty or it is a pair of a $\tau$ and a $\tau$ list
  - $\tau$ list $= \text{unit} + (\tau \times \tau$ list$)$
  - This is a recursive equation. We take its solution to be the smallest set of values $L$ that satisfies the equation
    - $L = \{ \ast \} \cup (T \times L)$
    - Where $T$ is the set of values of type $\tau$
  - Another interpretation is that the recursive equation is taken up-to (modulo) set isomorphism

Recursive Types

- We introduce a recursive type constructor $\mu$ ("mu"):
  - $\mu t. \tau$
    - The type variable $t$ is bound in $\tau$
    - This stands for the solution to the equation $t \simeq \tau$ (i.e. $t$ is isomorphic with $\tau$)
    - Example: $\tau$ list $= \mu t. (\text{unit} + \tau \times t)$
  - This also allows "unnamed" recursive types
- We introduce syntactic (sugary) operations for the conversion between $\mu t. \tau$ and $[\mu t. \tau]/t$
  - e.g. between "$\tau$ list" and "unit + ($\tau \times \tau$ list)"
  - $e ::= \ldots \mid \text{fold}_{\mu t. \tau} e \mid \text{unfold}_{\mu t. \tau} e$
  - $\tau ::= \ldots \mid t \mid \mu t. \tau$

Type Rules for Recursive Types

- $\Gamma \vdash e : \mu t. \tau$
  - $\Gamma \vdash \text{unfold}_{\mu t. \tau} e : [\mu t. \tau]/t \tau$
  - $\Gamma \vdash e : [\mu t. \tau]/t \tau$
  - $\Gamma \vdash \text{fold}_{\mu t. \tau} e : \mu t. \tau$
- The typing rules are syntax directed
- Often, for syntactic simplicity, the fold and unfold operators are omitted
  - This makes type checking somewhat harder

Example with Recursive Types

- Lists
  - $\tau$ list $= \mu t. (\text{unit} + \tau \times t)$
  - $\text{nil}_\tau = \text{fold}_{\mu t. \tau} (\text{injl } \ast)$
  - $\text{cons}_\tau = \lambda x: \tau. \lambda L: \tau$ list. $\text{fold}_{\mu t. \tau} \text{injr} (x, L)$
  - A list length function
    - $\text{length}_\tau = \lambda L: \tau$ list. $\text{case } (\text{unfold}_{\mu t. \tau} L)\text{ of } \lambda \text{injl } x \Rightarrow 0\text{ } | \lambda \text{injr } y \Rightarrow 1 + \text{length}_\tau (\text{snd } y)$
    - (At home ...) Verify that
      - $\text{nil}_\tau :: \tau$ list
      - $\text{cons}_\tau :: \tau \rightarrow \tau$ list $\rightarrow \tau$ list
      - $\text{length}_\tau :: \tau$ list $\rightarrow \text{int}$

Dynamics of Recursive Types

- We add a new form of values
  - $v ::= \ldots \mid \text{fold}_{\mu t. \tau} v$
  - The purpose of fold is to ensure that the value has the recursive type and not its unfolding
- The evaluation rules:
  - $e \Downarrow v$ \quad $e \Downarrow \text{fold}_{\mu t. \tau} v$
  - The folding annotations are for type checking only
  - They can be dropped after type checking
Recursive Types in ML
- The language ML uses a simple syntactic trick to avoid having to write the explicit fold and unfold.
- In ML recursive types are bundled with union types:
  \[ \text{type } t = C_1 \text{ of } \tau_1 \; | \; C_2 \text{ of } \tau_2 \; | \; \ldots \; | \; C_n \text{ of } \tau_n \] (* t can appear in \( \tau_i \))
  - e.g., "type intlist = Nil of unit | Cons of int * intlist"
- When the programmer writes \( \text{Cons } (5, l) \) - the compiler treats it as \( \text{fold}_{\text{intlist}} (\text{injlr } (5, l)) \)
- When the programmer writes
  - case \( e \) of Nil ⇒ … | Cons (h, t) ⇒ …
  the compiler treats it as
  - case unfold_{\text{intlist}} \( e \) of Nil ⇒ … | Cons (h,t) ⇒ …

Encoding Call-by-Value \( \lambda \)-calculus in \( F_1^\mu \)
- So far, \( F_1 \) was so weak that we could not encode non-terminating computations
  - Cannot encode recursion
  - Cannot write the \( \lambda x.x \) (self-application)
- The addition of recursive types makes typed \( \lambda \)-calculus as expressive as untyped \( \lambda \)-calculus!
- We could show a conversion algorithm from call-by-value untyped \( \lambda \)-calculus to call-by-value \( F_1^\mu \)

Untyped Programming in \( F_1^\mu \)
- We write \( e \) for the conversion of the term \( e \) to \( F_1^\mu \)
  - The type of \( e \) is \( V = \mu t. t \rightarrow t \)
- The conversion rules
  \[
  \begin{align*}
  x & \Rightarrow x \\
  \lambda x. e & \Rightarrow \text{fold}_\mu (\lambda x:V. e) \\
  e_1 e_2 & \Rightarrow (\text{unfold}_\mu e_1) e_2 \\
  \end{align*}
  \]
- Verify that
  1. \( \vdash e : V \)
  2. \( e \downarrow v \) if and only if \( e \downarrow^* v \)
- We can express non-terminating computation
  \[
  D = (\text{unfold}_\mu (\text{fold}_\mu (\lambda x:V. (\text{unfold}_\mu x) x))) (\text{fold}_\mu (\lambda x:V. (\text{unfold}_\mu x) x))
  \]
  or, equivalently
  \[
  D = (\lambda x:V. (\text{unfold}_\mu x) x) (\text{fold}_\mu (\lambda x:V. (\text{unfold}_\mu x) x))
  \]

Homework
- Read Goodenough article
  - Optional, perspectives on exceptions
- Work on Homework 5!
- Work on your projects!
  - Status Update Due Soon