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 `exn` of exceptions
  - `exn` could be int to model error codes
    - In Java or C++, `exn` is a special object type
  * Supposedly.

Modeling Exceptions

- Syntax
  ```
  e ::= ... | raise e | try e handle x ⇒ e2
  t ::= ... | exn
  
  We ignore here how exception values are created
  - In examples we will use integers as exception values
  - The handler binds `x` in `e2` to the actual exception value
  - The “`raise`” expression never returns to the immediately enclosing context
    - `1 + raise 2` is well-typed
    - if (raise 2) then 1 else 2 is also well-typed
    - (raise 2) 5 is also well-typed
    - What should be the type of `raise`?
  ```

Example with Exceptions

- A (strange) factorial function
  ```
  let f = λx:int.λres:int. if x = 0 then raise res else f (x - 1) (res * x)
  ```
  in try f 5 1 handle x ⇒ 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
  ```
  Γ ⊢ e : exn
  Γ ⊢ raise e : τ
  Γ ⊢ e1 : τ, x : exn ⊢ e2 : τ
  Γ ⊢ try e1 handle x ⇒ e2 : τ
  ```
  - 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 | 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
Contexts for Exceptions

- **Contexts**
  - H ::= • | H e | v H | raise H | try H handle x ⇒ e

- **Propagating contexts**
  - Contexts that propagate exceptions to their own enclosing contexts
  - P ::= • | P e | v P | 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[λx:τ e) v] (normal lambda calculus)
    - H[try v handle x ⇒ e] (handle it or not)
    - H[try P[raise v] handle x ⇒ e] (propagate!)
    - P[raise v] (uncaught exception)

Contextual Semantics for Exceptions

- **Small-step reduction rules**
  - H[(λx:τ e) v] → H[[v/x] e]
  - H[try v handle x ⇒ e] → H[v]
  - H[try P[raise v] handle x ⇒ e] → H[[v/x] e]
  - P[raise v] → 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)

Continuations

- Syntax:
  - Syntax: e ::= callcc k in e | throw e, e₂
  - τ ::= • | τ cont

- The type of a continuation that expects a τ
- 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, e₂ - evaluates e to a continuation, e₂ to a value and invokes the continuation with the value of e₂ (just wait, we’ll explain it!)

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
Example with Continuations

- Example: another strange factorial
  \texttt{callcc k in}
  
  \texttt{let f = \lambda x:int. \lambda res:int. if x = 0 then throw k res}
  \texttt{else f (x - 1) (x \ast 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 \textit{strictly more powerful} than exceptions
  - The destination is not tied to the call stack

Static Semantics of Continuations

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

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

Dynamic Semantics of Continuations

- Use \textit{contextual semantics} (wow, again!)
  - Contexts are now manipulated directly
  - Contexts are values of \textit{type} \( \tau \cont \)
- Contexts
  \[
  H ::= \bullet \mid H \ e \mid v \ H \mid \text{throw } H, e_1 \mid \text{throw } v, H_2
  \]
- Evaluation rules
  - \( H[(\lambda x.e) \ v] \rightarrow H[\[v/x\] e] \)
  - \( H[\text{callcc } k \ in \ e] \rightarrow H[\text{callcc } v/k \ e] \)
  - \( H[\text{throw } H, v] \rightarrow H[v] \)
- callcc duplicates the current continuation
- Note that throw abandons its own context

Implementing Coroutines with Continuations

- Example:
  \texttt{let client = \lambda k. let res = callcc k' in throw k k' in}
  \texttt{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)

  \texttt{let getnext =}
  \texttt{\lambda L. \lambda 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 \textit{continuation saves the entire context}: program counter, local variables, call stack, and the heap!
- In actual implementations the \textit{heap is not saved!}
- Saving the stack is done with various tricks, but it is \textit{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 \textit{never returns directly}: instead the function is \textit{given a continuation to invoke with its result}.
- Instead of
  \texttt{f(int a) \{ return h(g(e)); \}}
  \texttt{we write}
  \texttt{f(int a, cont k) \{ g(e, \lambda r. h(r, k)) \}}
- Advantages:
  - interesting compilation scheme (supports callcc easily)
  - \textit{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 \( e ::= x | n | e_1 + e_2 | \text{if } e_1 \text{ then } e_2 \text{ else } e_3 \)
- \( \lambda x.e \mid e_1 e_2 \)
- Define \( \text{cps}(e, k) \) as the code that computes \( e \) in CPS and passes the result to continuation \( k \)
- \( \text{cps}(x, k) = k x \)
- \( \text{cps}(n, k) = k n \)
- \( \text{cps}(e_1 + e_2, k) = \text{cps}(e_1, \lambda n_1. \text{cps}(e_2, \lambda n_2. k (n_1 + n_2))) \)
- \( \text{cps}(\lambda x.e, k) = k (\lambda x. k' . \text{cps}(e)) \)
- \( \text{cps}(e_1 e_2, k) = \text{cps}(e_1, \lambda f_1. \text{cps}(e_2, \lambda v_2. f_1 v_2 k)) \)
- Example: \( \text{cps} (\text{h}(\text{g}(5)), k) = \text{g}(5, \lambda x. \text{h} x k) \)
- Notice the order of evaluation being explicit

Recursive Types: Lists

- We want to define recursive data structures
- Example: \textbf{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
  - \( \tau \text{ list} = \text{unit} + (\tau \times \tau \text{ list}) \)
  - This is a recursive equation. We take its solution to be the smallest set of values \( L \) that satisfies the equation
  \[
  L = \{*\} \cup (T \times L)
  \]
  where \( T \) is the set of values of type \( \tau \)
  - Another interpretation is that the recursive equation is taken up-to (modulo) set isomorphism

Type Rules for Recursive Types

- The typing rules are syntax directed
- Often, for syntactic simplicity, the fold and unfold operators are omitted
  - This makes type checking somewhat harder

Dynamics of Recursive Types

- We add a new form of values
  \( v ::= ... \mid \text{fold}_{\mu \tau} v \)
  - The purpose of fold is to ensure that the value has the recursive type and not its unfolding
- The evaluation rules:
  \( e \Downarrow v \quad e \Downarrow \text{fold}_{\mu \tau} v \)
- 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.
  - E.g., "type intlist = Nil of unit | Cons of int * intlist".
- When the programmer writes `Cons (5, l)` the compiler treats it as `fold ( unfold (injlr (5, l)))`.
- When the programmer writes `case e of Nil ⇒ ... | Cons (h, t) ⇒ ...` the compiler treats it as `case unfold e of Nil ⇒ ... | Cons (h, t) ⇒ ...`.

Encoding Call-by-Value \(\lambda\)-calculus in \(F^1_\mu\)

- So far, \(F^1_\mu\) was so weak that we could not encode non-terminating computations.
  - Cannot encode recursion.
  - Cannot write the \(\lambda x.x\ x\) (self-application).
- The addition of recursive types makes typed \(\lambda\)-calculus as expressive as untyped \(\lambda\)-calculus!
- We can 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
  \[
  \begin{align*}
  x & \rightarrow x \\
  \lambda x. e & \rightarrow \text{fold}(\lambda x:V. e) \\
  e_1 e_2 & \rightarrow \text{fold} (\text{unfold} e_1) e_2 \\
  \end{align*}
  \]
- Verify that
  1. \(\vdash e : V\)
  2. \(e \downarrow v\) if and only if \(e \downarrow v\)
- We can express non-terminating computation
  \[
  D = \text{fold} (\text{unfold} (\lambda x:V. (\text{unfold} x) x)) (\text{fold} (\lambda x:V. (\text{unfold} x) x)))
  \]
  or, equivalently
  \[
  D = (\lambda x:V. (\text{unfold} x) x) (\text{fold} (\lambda x:V. (\text{unfold} x) x)))
  \]

Homework

- Read Goodenough article
  - Optional, perspectives on exceptions
- Thursday: Class Survey #2
- Work on your projects!
  - Status Update Due: Thursday