Types for Continuations and Recursive Types

Meeting 22, CSCI 5535, Spring 2009

One-Slide Summary

- Exceptions are like non-local gotos; they are used to propagate errors. We will use contextual semantics to model them.
- Continuations allow you to take a snapshot of the current execution and store it for later use. They are often used for threads or backtracking. We will use contextual semantics to model them.

Example with Exceptions

A (strange) factorial function

\[
\text{let } f = \lambda x:\text{int}.\lambda res:\text{int}. \begin{cases} 
\text{if } x = 0 \text{ then raise res} \\
\text{else } f (x - 1) (res \times x) 
\end{cases}
\]

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

Continuations

- 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!)

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

Exceptions and Continuations
Continuations

\[ e ::= \text{letcc } k \text{ in } e \mid \text{throw } e \]

\[ \tau ::= \tau \mid \text{cont} \]

- \( \tau \text{ cont} \) - the type of a continuation that expects a \( \tau \)
- \( \text{letcc } k \text{ in } e \) - sets \( k \) to the current context of the execution and then evaluates expression \( e \)
  - when \( e \) terminates, the whole letcc terminates
  - \( e \) can invoke the saved continuation (many times even)
  - \( k \) is bound in \( e \)
  - sometimes called callcc
- \( \text{throw } e \) - evaluates \( e \) 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

```latex
\text{letcc } k \text{ in } \lambda x: \text{int}. \lambda \text{res: int}. \text{if } x = 0 \text{ then } \text{throw } k \text{ res } \\
\text{else } f (x - 1) (x \times \text{res}) \\
\text{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 letcc 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

\[ \Gamma, k : \tau \text{ cont} \vdash e : \tau \]

\[ \begin{align*}
\Gamma \vdash \text{letcc } k \text{ in } e : \tau \\
\Gamma, k : \tau \text{ cont} \vdash e : \tau \\
\Gamma \vdash \text{throw } e_1 e_2 : \tau'
\end{align*} \]

- Note the result of letcc is of type \( \tau \)
- "letcc \( k \) in \( e \)" returns in two possible situations
  - \( e \) throws to \( k \) a value of type \( \tau \), or
  - \( e \) 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

\[ H ::= \epsilon \mid H e \mid v H \mid \text{throw } H_1 e_2 \mid \text{throw } v_1 H_2 \]

- Evaluation rules

\[ \begin{align*}
H[(\lambda x. e) v] & \rightarrow H[e[v/x]] \\
H[\text{letcc } k \text{ in } e] & \rightarrow H[e/k] \\
H[\text{throw } H_1 v_2] & \rightarrow H[v_2]
\end{align*} \]

- letcc duplicates the current continuation
- Note that throw abandons its own context
Implementing Coroutines with Continuations

- Example:
  let client = \k. let res = (letcc 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)

  getnext [0;1;2;3;4;5] (letcc 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)

Bonus: 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
  ```
  f(int a) { return h(g(e)); }
  ```
- we write
  ```
  f(int a, cont k) { g(e, \r. h(r, k)) }
  ```

- Advantages:
  - interesting compilation scheme (supports letcc 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 | n | e1 + e2 | if e1 then e2 else e3
  | \x.e | e1 e2
- 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(e1 + e2, k) =
    cps(e1, \n1. cps(e2, \n2. k (n1 + n2)))
  - cps(\x.e, k) = k (\x.xk). cps(e, k))
  - cps(e1, e2, k) = cps(e1, \k. f1 cps(e2, \k. f2 v2 k))

  - Example: cps (h(g(5)), k) = g(5, \x.h x k)

  - Notice the order of evaluation being explicit

Recursive Types
One-Slide Summary

• Recursive types (e.g., τ list) make the typed lambda calculus as powerful as the untyped lambda calculus.

Recursive Types: Lists

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

Recursive Types

• We introduce a recursive type constructor \( \mu (\mu) : \)

  \[ \mu t. τ \]
  - The type variable \( t \) is bound in \( τ \)
  - This stands for the solution to the equation
  \[ t \equiv τ \]
  - Example: \( τ \text{ list} = \mu t. (\text{unit} + \tau \times \tau \text{ list}) \)
  - This also allows "unnamed" recursive types
• We introduce syntactic (sugary) operations for the conversion between \( \mu t. τ \) and \( (\mu t. τ)/t \):
  - e ::= … | fold\( _{\mu t. τ} \) e | unfold\( _{\mu t. τ} \) e
  \[ τ ::= … | t | \mu t. τ \]

Example with Recursive Types

• Lists

  \[ τ \text{ list} = \mu t. (\text{unit} + τ \times τ) \]

  \[ \text{nil}_τ = \text{fold}_{μ τ} (\text{injl } *) \]

  \[ \text{cons}_τ = λx: τ. λL: τ \text{ list}. \text{fold}_{μ τ} \text{injr } (x, L) \]

• A list length function

  \[ \text{length}_τ = λL: τ \text{ list}. \text{case } (\text{unfold}_{\mu τ} \text{injL } x) \Rightarrow \begin{cases} 0 & \text{injl } x \Rightarrow 0 \text{ length}_τ, (\text{and } y) \\ 1 + \text{length}_τ & \text{injr } y \Rightarrow 1 + \text{length}_τ, (\text{and } y) \end{cases} \]

• (At home ...) Verify that
  - nil\( _{τ} : τ \text{ list} \)
  - cons\( _{τ} : τ \rightarrow τ \text{ list} \rightarrow τ \text{ list} \)
  - length\( _{τ} : τ \text{ list} \rightarrow \text{int} \)

Type Rules for Recursive Types

\[ \Gamma; e: \mu t. τ \兴起 \]

\[ \Gamma \vdash \text{unfold}_{\mu t. τ} e : [\mu t. τ]/t \]
\[ \Gamma \vdash \text{fold}_{\mu t. τ} e : \mu t. τ \]
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

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:
  
  \[ e \Downarrow v \]
  \[ e \Downarrow \text{fold}_{\mu t. \tau} v \]
  \[ \text{fold}_{\mu t. \tau} e \Downarrow \text{fold}_{\mu t. \tau} v \]
  \[ \text{unfold}_{\mu t. \tau} e \Downarrow v \]

  - The folding annotations are for type checking only
  - They can be dropped after type checking

Recursive Types in ML

- ML uses a simple syntactic trick to avoid having to write the explicit fold and unfold
- In ML recursive types are bundled with sum types
  
  \[ \text{type } t \equiv C_1 \text{ of } \tau_1 \mid C_2 \text{ of } \tau_2 \mid \ldots \mid C_n \text{ of } \tau_n \]
  (* \( t \) can appear in \( \tau \)*)
  
  - e.g., "type intlist = Nil of unit | Cons of int * intlist"

- When the programmer writes \( \text{Cons} (5, l) \)
  - the compiler treats it as \( \text{fold}_{\text{intlist}} \text{ (injr } (5, l)) \)

- When the programmer writes
  
  - \( \text{case } e \text{ of } \text{Nil } \Rightarrow \ldots \mid \text{Cons } (h, t) \Rightarrow \ldots \)
  
  the compiler treats it as

  - \( \text{case } \text{unfold}_{\text{intlist}} e \text{ of } \text{Nil } \Rightarrow \ldots \mid \text{Cons } (h, t) \Rightarrow \ldots \)

Encoding Call-by-Value \( \lambda \text{-calculus in } F_{1^\mu} \)

- 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 \( F_{1^\mu} \)

Subtyping Recursive Types

- Recall \( \tau \text{ list } \equiv \mu t. (\text{unit + } t \times t) \)
  - We would like \( \tau \text{ list } \equiv \sigma \text{ list } \) whenever \( \tau \equiv \sigma \)
- Covariance?
  
  \[ \tau \equiv: \sigma \quad \mu t. \tau \equiv: \mu t. \sigma \]
  
  Wrong!

- This is wrong if \( t \) occurs contravariantly in \( \tau \)
- Take \( \tau \equiv \mu t. t \rightarrow \text{int} \) and \( \sigma \equiv \mu t. t \rightarrow \text{real} \)
- Above rule says that \( \tau \equiv: \sigma \)
- We have \( t \rightarrow \text{int} \equiv \sigma \rightarrow \text{real} \)
- \( \tau \equiv: \sigma \) would mean covariant function type!
- How can we get safe subtyping for lists?

Subtyping Recursive Types

- The correct rule
  
  \[ \tau \equiv: \sigma \quad \mu t. \tau \equiv: \mu s. \sigma \]

- We add as an assumption that the type variables stand for types with the desired subtype relationship
  - Before we assumed they stood for the same type!
- Verify that now subtyping works properly for lists
- There is no subtyping between \( \mu t. t \rightarrow \text{int} \) and \( \mu t. t \rightarrow \text{real} \)