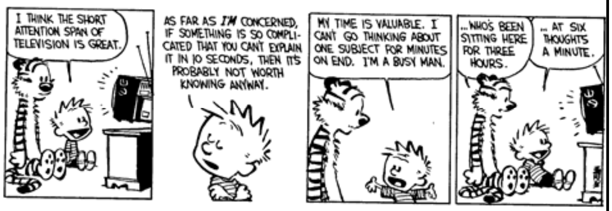
Type Systems For: Exceptions, Continuations, and Recursive Types



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 types

** Supposedly.*

Modeling Exceptions

- Syntax
  - e ::= … | raise e | try e_1 handle x ⇒ e_2
  - τ ::= … | 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 "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

$$\frac{\Gamma \vdash e : \texttt{exn}}{\Gamma \vdash \texttt{raise}\ e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \qquad \Gamma, x : \texttt{exn} \vdash e_2 : \tau}{\Gamma \vdash \texttt{try}\ e_1\ \texttt{handle}\ x \Longrightarrow e_2 : \tau}$$

- 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 | 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 :: = • | 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

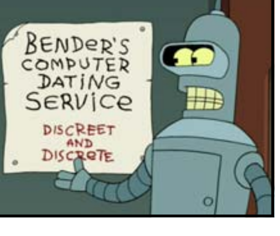
- The addition of exceptions preserves type soundness
- Exceptions are like *non-local goto*
- However, they cannot be used to implement recursion
  - Thus we *still* cannot write (well-typed) non-terminating programs
- There are a number of ways to implement exceptions (e.g., "zero-cost" exceptions)

Continuations

- 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_1 = C [• + e_2]
    - Alternatively, as λx_1. C [$x_1 + e_2$]
  - When we finish evaluating e_1 to v_1, we fill the context and continue with C[$v_1 + e_2$]
  - But the C_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 ::= callcc k in e | throw e_1 e_2

  τ ::= … | τ cont
- 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_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 = λx:int.λres: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* that exceptions
  - The destination is not tied to the call stack

Static Semantics of Continuations

$$\frac{\Gamma, k : \tau \ \texttt{cont} \vdash e : \tau}{\Gamma \vdash \texttt{callcc}\ k\ \texttt{in}\ e : \tau}$$

$$\frac{\Gamma \vdash e_1 : \tau \ \texttt{cont} \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash \texttt{throw}\ e_1\ e_2 : \tau'}$$

- Note that the result of callcc is of type τ
  "callcc k in e" returns in two possible situations
  1. e *throws* to k a value of type τ, or
  2. e *terminates normally* with a value of type τ
- Note that throw has any type τ'
  - 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 τ cont
- Contexts
  H ::= • | H e | v H | throw H_1 e_2 | throw v_1 H_2
- Evaluation rules
  - **H[(λx.e) v]** → **H[[v/x] e]**
  - **H[callcc k in e]** → **H[[H/k] e]**
  - **H[throw H_1 v_2]** → **H_1[v_2]**
- callcc duplicates the current continuation
- Note that throw abandons its own context

Implementing Coroutines with Continuations

- Example:

```
let client = λk. 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 =
    λL.λk. 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 `f(int a) { return h(g(e); }`
- we write `f(int a, cont k) { g(e, λr. h(r, k) ) }`
- 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

  $cps(x, k) = k\ x$

  $cps(n, k) = k\ n$

  $cps(e_1 + e_2, k) =$
  $\quad cps(e_1, \lambda n_1.cps(e_2, \lambda n_2.k\ (n_1 + n_2)))$

  $cps(\lambda x.e, k) = k\ (\lambda x \lambda k'.\ cps(e, k'))$

  $cps(e_1\ e_2, k) = cps(e_1, \lambda f_1.cps(e_2, \lambda v_2.\ f_1\ v_2\ k))$
- Example: cps (h(g(5)), k) = g(5, λx.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 τ (a τ list) is *either* empty *or* it is a pair of a τ and a τ list

    τ list = unit + (τ × τ 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 τ
  - Another interpretation is that the recursive equation is taken up-to (modulo) set isomorphism

Recursive Types

- We introduce a recursive type constructor μ ("mu"):

  μt. τ
  - The type variable t is bound in τ
  - This stands for the solution to the equation

    $t \simeq \tau$ (t is isomorphic with τ)
  - Example: **τ list = μt. (unit + τ × t)**
  - This also allows "unnamed" recursive types
- We introduce syntactic (sugary) operations for the conversion between $\mu t.\tau$ and $[\mu t.\tau/t]\tau$
- e.g. between "τ list" and "unit + (τ × τ list)"

  $e ::= \ldots \quad \mid \text{fold}_{\mu t.\tau}\ e \mid \text{unfold}_{\mu t.\tau}\ e$

  $\tau ::= \ldots \quad \mid t \mid \mu t.\tau$

Example with Recursive Types

- Lists

  τ list = μt. (unit + τ × t)

  nil_τ = $\text{fold}_{\tau\ \text{list}}$ (injl *)

  cons_τ = λx:τ.λL:τ list. $\text{fold}_{\tau\ \text{list}}$ injr (x, L)
- A list length function

  length_τ = λL:τ list.
  case ($\text{unfold}_{\tau\ \text{list}}$ L) of injl x ⇒ 0
  | injr y ⇒ 1 + length_τ (snd y)
- (At home ...) Verify that
  - nil_τ : τ list
  - cons_τ : τ → τ list → τ list
  - length_τ : τ list → int

Type Rules for Recursive Types

$$\frac{\Gamma \vdash e : \mu t.\tau}{\Gamma \vdash \texttt{unfold}_{\mu t.\tau}\ e : [\mu t.\tau/t]\tau}$$

$$\frac{\Gamma \vdash e : [\mu t.\tau/t]\tau}{\Gamma \vdash \texttt{fold}_{\mu t.\tau}\ e : \mu t.\tau}$$

- 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:

$$\frac{e \Downarrow v}{\texttt{fold}_{\mu t.\tau}\ e \Downarrow \texttt{fold}_{\mu t.\tau}\ v} \qquad \frac{e \Downarrow \texttt{fold}_{\mu t.\tau}\ v}{\texttt{unfold}_{\mu t.\tau}\ e \Downarrow v}$$

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

Recursive Types in ML

- 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*
  - **type t = C_1 of τ_1 | C_2 of τ_2 | ... | C_n of τ_n (* t can appear in τ_i *)**
  - e.g., "type intlist = Nil of unit | Cons of int * intlist"
- When the programmer writes Cons (5, l)
  - the compiler treats it as $\text{fold}_{\text{intlist}}$ (injlr (5, l))
- When the programmer writes
  - case e of Nil ⇒ ... | Cons (h, t) ⇒ ...

  the compiler treats it as
  - case $\text{unfold}_{\text{intlist}}$ e of Nil ⇒ ... | Cons (h,t) ⇒ ...

#25

Encoding Call-by-Value λ-calculus in F_1^{μ}

- So far, F_1 was so weak that we could not encode non-terminating computations
  - Cannot encode recursion
  - Cannot write the λx.x x (self-application)
- The addition of recursive types makes typed λ-calculus *as expressive as untyped λ-calculus*!
- We could show a conversion algorithm from call-by-value untyped λ-calculus to call-by-value F_1^{μ}

#26

Untyped Programming in F_1^{μ}

- We write $\underline{e}$ for the conversion of the term e to F_1^{μ}
  - The type of $\underline{e}$ is V = μt. t → t
- The conversion rules

  $\underline{x}$ = x

  $\underline{\lambda x.\ e}$ = fold_V (λx:V. $\underline{e}$)

  $\underline{e_1\ e_2}$ = (unfold_V $\underline{e_1}$) $\underline{e_2}$
- Verify that
  1. $\cdot \vdash \underline{e} : V$
  2. e ⇓ v if and only if $\underline{e}$ ⇓ $\underline{v}$
- We can express non-terminating computation

  D = (unfold_V (fold_V (λx:V. (unfold_V x) x))) (fold_V (λx:V. (unfold_V x) x)))

  or, equivalently

  D = (λx:V. (unfold_V x) x) (fold_V (λx:V. (unfold_V x) x)))

#27

Homework

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

#28