Type Systems

Lecture 11: Applications of Continuations, and Dependent Types

Neel Krishnaswami
University of Cambridge
Applications of Continuations
Applications of Continuations

We have seen that:

- Classical logic has a beautiful inference system
- Embeds into constructive logic via double-negation translations
- This yields an operational interpretation
- What can we program with continuations?
The Typed Lambda Calculus with Continuations

Types  \[ X ::= 1 \mid X \times Y \mid 0 \mid X + Y \mid X \rightarrow Y \mid \neg X \]

Terms  \[ e ::= x \mid \langle \rangle \mid \langle e, e \rangle \mid \text{fst } e \mid \text{snd } e \]
\[ \mid \text{abort} \mid L_e \mid R_e \mid \text{case}(e, L x \rightarrow e', R y \rightarrow e'') \]
\[ \mid \lambda x : X. e \mid e e' \]
\[ \mid \text{throw}(e, e') \mid \text{letcont } x. e \]

Contexts  \[ \Gamma ::= \cdot \mid \Gamma, x : X \]
Continuation Typing

\[
\frac{\Gamma, u : \neg X \vdash e : X}{\Gamma \vdash \text{letcont } u : \neg X. e : X} \quad \text{CONT}
\]

\[
\frac{\Gamma \vdash e : \neg X \quad \Gamma \vdash e' : X}{\Gamma \vdash \text{throw}_Y(e, e') : Y} \quad \text{THROW}
\]
Continuation API in Standard ML

```
signature CONT = sig
  type 'a cont
  val callcc : ('a cont -> 'a) -> 'a
  val throw : 'a cont -> 'a -> 'b
end
```

<table>
<thead>
<tr>
<th>SML</th>
<th>Type Theory</th>
</tr>
</thead>
<tbody>
<tr>
<td>'a cont</td>
<td>¬A</td>
</tr>
<tr>
<td>throw k v</td>
<td>throw(k, v)</td>
</tr>
<tr>
<td>callcc (fn x =&gt; e)</td>
<td>letcont x : ¬X. e</td>
</tr>
</tbody>
</table>
An Inefficient Program

```
val mul : int list -> int

fun mul [] = 1
    | mul (n :: ns) = n * mul ns

• This function multiplies a list of integers
• If 0 occurs in the list, the whole result is 0
```
val mul' : int list -> int

fun mul' [] = 1
  | mul' (0 :: ns) = 0
  | mul' (n :: ns) = n * mul ns

• This function multiplies a list of integers
• If 0 occurs in the list, it immediately returns 0
  • mul' [0,1,2,3,4,5,6,7,8,9] will immediately return
  • mul' [1,2,3,4,5,6,7,8,9,0] will multiply by 0, 9 times
val loop : int cont -> int list -> int = fn

fun loop return [] = 1

| loop return (0 :: ns) = throw return 0

| loop return (n :: ns) = n * loop return ns

val mul_fast : int list -> int = fn

fun mul_fast ns = callcc (fn ret => loop ret ns)

• loop multiplies its arguments, unless it hits 0
• In that case, it throws 0 to its continuation
• mul_fast captures its continuation, and passes it to loop
• So if loop finds 0, it does no multiplications!
McCarthy’s amb Primitive

- In 1961, John McCarthy (inventor of Lisp) proposed a language construct \( \text{amb} \)
- This was an operator for \textit{angelic nondeterminism}

```plaintext
let val x = amb [1,2,3]
val y = amb [4,5,6]
in
assert (x * y = 10);
(x, y)
end
(* Returns (2,5) *)
```

- Does search to find a successful assignment of values
- Can be implemented via backtracking – \textit{using continuations}
The AMB signature

```
signature AMB = sig
  (* Internal implementation *)
val stack : int option option cont list ref
val fail : unit -> 'a

  (* External API *)
exception AmbFail
val assert : bool -> unit
val amb : int list -> int
end
```
exception AmbFail

val stack : int option option cont list ref = ref []

fun fail () =
    case !stack of
    [] => raise AmbFail
    | (k :: ks) => (stack := ks; throw k NONE)

fun assert b =
    if b then () else fail()
fun amb [] = fail ()
| amb (x :: xs) =
  let fun next y k =
    (stack := k :: !stack;
     SOME y)
  in
    case callcc (next x) of
    SOME v => v,
    | NONE => amb xs.
  end
fun test2() = 
  let val x = amb [1,2,3,4,5,6] 
  val y = amb [1,2,3,4,5,6] 
  val z = amb [1,2,3,4,5,6] 
  in 
    assert(x + y + z >= 13); 
    assert(x > 1); 
    assert(y > 1); 
    assert(z > 1); 
    (x, y, z) 
  end 
  (* Returns (2, 5, 6) *)
Conclusions

- **amb** required the *combination* of state and continuations
- Theorem of Andrzej Filinski that this is *universal*
- Any “definable monadic effect” can be expressed as a combination of state and first-class control:
  - Exceptions
  - Green threads
  - Coroutines/generators
  - Random number generation
  - Nondeterminism
Dependent Types
### The Curry Howard Correspondence

<table>
<thead>
<tr>
<th>Logic</th>
<th>Language</th>
</tr>
</thead>
<tbody>
<tr>
<td>Intuitionistic Propositional Logic</td>
<td>STLC</td>
</tr>
<tr>
<td>Classical Propositional Logic</td>
<td>STLC + 1(^{st}) class continuations</td>
</tr>
<tr>
<td>Pure Second-Order Logic</td>
<td>System F</td>
</tr>
</tbody>
</table>

- Each logical system has a corresponding computational system
- One thing is missing, however
- Mathematics uses quantification over *individual elements*
- Eg, \(\forall x, y, z, n \in \mathbb{N}. \text{if } n > 2 \text{ then } x^n + y^n \neq z^n\)
A Logical Curiosity

\[ \frac{\Gamma \vdash z : \mathbb{N}}{\mathbb{N} \vdash z} \]

\[ \frac{\Gamma \vdash e : \mathbb{N}}{\mathbb{N} \vdash s(e)} \]

\[ \frac{\Gamma \vdash e_0 : \mathbb{N} \quad \Gamma \vdash e_1 : X \quad \Gamma, x : X \vdash e_2 : X}{\Gamma \vdash \text{iter}(e_0, z \rightarrow e_1, s(x) \rightarrow e_2) : X} \]

\[ \mathbb{N} \] is the type of natural numbers

- Logically, it is equivalent to the unit type:
  - \((\lambda x : 1. z) : 1 \rightarrow \mathbb{N}\)
  - \((\lambda x : \mathbb{N}. \langle \rangle) : \mathbb{N} \rightarrow 1\)

- Language of types has no way of distinguishing \(z\) from \(s(z)\).
Dependent Types

- Language of types has no way of distinguishing $z$ from $s(z)$.
- So let’s fix that: let types refer to values
- Type grammar and term grammar mutually recursive
- Huge gain in expressive power
• Much of earlier course leaned on prior knowledge of ML for motivation
• Before we get to the theory of dependent types, let’s look at an implementation
• Agda: a dependently-typed functional programming language
• http://wiki.portal.chalmers.se/agda/pmwiki.php
data Bool : Set where
  true : Bool
  false : Bool

not : Bool → Bool
not true = false
not false = true

• Datatype declarations give constructors and their types
• Functions given type signature, and clausal definition
Agda: Inductive Datatypes

```agda
data Nat : Set where
  z : Nat
  s : Nat → Nat

_+_ : Nat → Nat → Nat
z + m = m
s n + m = s (n + m)

_×_ : Nat → Nat → Nat
z × m = z
s n × m = m + (n × m)
```

- Datatype constructors can be recursive
- Functions can be recursive, but checked for termination
data List (A : Set) : Set where

  [] : List A
  _,_ : A → List A → List A

app : (A : Set) → List A → List A → List A
app A [] ys = ys
app A (x , xs) ys = x , app A xs ys

app' : {A : Set} → List A → List A → List A
app' [] ys = ys
app' (x , xs) ys = (x , app' xs ys)

- Datatypes can be polymorphic
- app has F-style explicit polymorphism
- app' has implicit, inferred polymorphism
data Vec (A : Set) : Nat -> Set where

[] : Vec A z
_,_ : {n : Nat} -> A -> Vec A n -> Vec A (s n)

• This is a length-indexed list
• Cons takes a head and a list of length \( n \), and produces a list of length \( n + 1 \)
• The empty list has a length of 0
Agda: Indexed Datatypes

```agda
data Vec (A : Set) : Nat → Set where
  [] : Vec A z
  _,_ : {n : Nat} → A → Vec A n → Vec A (s n)

head : {A : Set} → {n : Nat} → Vec A (s n) → A
head (x , xs) = x
```

- `head` takes a list of length > 0, and returns an element
- No `[]` pattern present
- Not needed for coverage checking!
- Note that `{n:Nat}` is *also* an implicit (inferred) argument
data Vec (A : Set) : Nat → Set where
  [] : Vec A z
  _,_ : {n : Nat} → A → Vec A n → Vec A (s n)

app : {A : Set} → {n m : Nat} →
    Vec A n → Vec A m → Vec A (n + m)
app [] ys = ys
app (x , xs) ys = (x , app xs ys)

• Note the appearance of \( n + m \) in the type
• This type guarantees that appending two vectors yields a vector whose length is the sum of the two
```agda
data Vec (A : Set) : Nat → Set where
  [] : Vec A z
  _,_ : {n : Nat} → A → Vec A n → Vec A (s n)

-- Won't typecheck!
app : {A : Set} → {n m : Nat} →
    Vec A n → Vec A m → Vec A (n + m)
app [] ys = ys
app (x , xs) ys = app xs ys
```

- We forgot to cons $x$ here
- This program won’t type check!
- Static typechecking ensures a runtime guarantee
The Identity Type

\[
\text{data } \equiv \{A : \text{Set} \} (a : A) : A \to \text{Set} \quad \text{where}
\]
\[
\text{refl} : a \equiv a
\]

- \(a \equiv b\) is the type of proofs that \(a\) and \(b\) are equal
- The constructor \text{refl} says that a term \(a\) is equal to itself
- Equalities arising from evaluation are automatic
- Other equalities have to be proved
An Automatic Theorem

\[
\text{data } _\equiv_ \{ A : \text{Set} \} (a : A) : A \to \text{Set} \text{ where }
\]
\[
\text{refl} : a \equiv a
\]

\[
_+_{\text{ Nat} \to \text{ Nat} \to \text{ Nat}}
\]
\[
z + m = m
\]
\[
s n + m = s (n + m)
\]

\[
z+-\text{-left-unit} : (n : \text{Nat}) \to (z + n) \equiv n
\]
\[
z+-\text{-left-unit} n = \text{refl}
\]

• z + n evaluates to n
• So Agda considers these two terms to be identical
A Manual Theorem

\[
\text{data } _\equiv_ \ \{A : \text{Set}\} \ (a : A) : A \to \text{Set where} \\
\text{refl} : a \equiv a
\]

\[
\text{cong} : \{A \ B : \text{Set}\} \to \{a \ a' : A\} \to \\
(f : A \to B) \to (a \equiv a') \to (f \ a \equiv f \ a')
\]

\[
\text{cong } f \ \text{refl} = \text{refl}
\]

\[
\text{z+-right-unit} : (n : \text{Nat}) \to (n + z) \equiv n
\]

\[
\text{z+-right-unit } z = \text{refl}
\]

\[
\text{z+-right-unit } (s \ n) = \text{cong } s \ (\text{z+-right-unit } n)
\]

• We prove the right unit law inductively
• Note that \textit{inductive proofs are recursive functions}
• To do this, we need to show that equality is a congruence
The Equality Toolkit

```plaintext
data _≡_ {A : Set} (a : A) : A → Set where
  refl : a ≡ a

sym : {A : Set} → {a b : A} →
  a ≡ b → b ≡ a
sym refl = refl

trans : {A : Set} → {a b c : A} →
  a ≡ b → b ≡ c → a ≡ c
trans refl refl = refl

cong : {A B : Set} → {a a' : A} →
  (f : A → B) → (a ≡ a') → (f a ≡ f a')
cong f refl = refl

• An equivalence relation is a reflexive, symmetric transitive relation
• Equality is congruent with everything
```
Commutativity of Addition

\[ z-+-\text{right} : (n : \text{Nat}) \rightarrow (n + z) \equiv n \]
\[ z-+-\text{right} \ z = \text{refl} \]
\[ z-+-\text{right} \ (s \ n) = \]
\[ \\quad \text{cong} \ s \ (z-+-\text{right} \ n) \]

\[ s-+-\text{right} : (n \ m : \text{Nat}) \rightarrow \]
\[ \quad (s (n + m)) \equiv (n + (s \ m)) \]
\[ s-+-\text{right} \ z \ m = \text{refl} \]
\[ s-+-\text{right} \ (s \ n) \ m = \]
\[ \quad \text{cong} \ s \ (s-+-\text{right} \ n \ m) \]

\[ +-\text{comm} : (i \ j : \text{Nat}) \rightarrow \]
\[ \quad (i + j) \equiv (j + i) \]
\[ +-\text{comm} \ z \ j = z-+-\text{right} \ j \]
\[ +-\text{comm} \ (s \ i) \ j = \text{trans} \ p2 \ p3 \]
\[ \text{where} \ p1 : (i + j) \equiv (j + i) \]
\[ p1 = +-\text{comm} \ i \ j \]
\[ p2 : (s (i + j)) \equiv (s (j + i)) \]
\[ p2 = \text{cong} \ s \ p1 \]
\[ p3 : (s (j + i)) \equiv (j + (s \ i)) \]
\[ p3 = s-+-\text{right} \ j \ i \]

- First we prove that adding zero on the right does nothing
- Then we prove that successor commutes with addition
- Then we use these two facts to inductively prove commutativity of addition
• Dependent types permit referring to program terms in types
• This enables writing types which state very precise properties of programs
  • Eg, equality is expressible as a type
• Writing a program becomes the same as proving it correct
• This is hard, like learning to program again!
• But also extremely fun...