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

$$
\begin{array}{llcl}
\text{Types} & X & ::= & 1 \mid X \times Y \mid 0 \mid X + Y \mid X \to Y \mid \neg X \\
\text{Terms} & e & ::= & x \mid \langle\rangle \mid \langle e, e \rangle \mid \mathsf{fst}\, e \mid \mathsf{snd}\, e \\
 & & \mid & \mathsf{abort} \mid \mathsf{L}\, e \mid \mathsf{R}\, e \mid \mathsf{case}(e, \mathsf{L}\, x \to e', \mathsf{R}\, y \to e'') \\
 & & \mid & \lambda x : X.\, e \mid e\, e' \\
 & & \mid & \mathsf{throw}(e, e') \mid \mathsf{letcont}\, x.\, e \\
\text{Contexts} & \Gamma & ::= & \cdot \mid \Gamma, x : X
\end{array}
$$

Continuation Typing

$$\frac{\Gamma, u : \neg X \vdash e : X}{\Gamma \vdash \mathrm{letcont}\, u : \neg X.\ e : X}\ \textsc{Cont} \qquad \frac{\Gamma \vdash e : \neg X \qquad \Gamma \vdash e' : X}{\Gamma \vdash \mathrm{throw}_Y(e, e') : Y}\ \textsc{Throw}$$

Continuation API in Standard ML

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

SML	Type Theory
`'a cont`	$\neg A$
`throw k v`	$\mathrm{throw}(k, v)$
`callcc (fn x => e)`	$\mathrm{letcont}\, x : \neg X.\ e$

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

A Less Inefficient Program

```
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

Even Less Inefficiency, via Escape Continuations

```
val loop = fn : int cont -> int list -> int
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
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 `amb`
- This was an operator for *angelic nondeterminism*

```
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 succesful assignment of values
- Can be implemented via backtracking – *using continuations*

The AMB signature

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

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

Implementation, Part 1

```
exception AmbFail
val stack
  : int 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()^^I^^I^^I
```

- `AmbFail` is the failure exception for unsatisfiable computations
- `stack` is a stack of backtrack points
- `fail` grabs the topmost backtrack point, and resumes execution there
- `assert` backtracks if its condition is false

Implementation, Part 2

```
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
```

- `amb []` backtracks immediately!
- `next y k` pushes `k` onto the backtrack stack, and returns `SOME y`
- Save the backtrack point, then see if we immediately return, or if we are resuming from a backtrack point and must try the other values

Examples

```
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

Logic	Language
Intuitionistic Propositional Logic	STLC
Classical Propositional Logic	STLC + 1^{st} class continuations
Pure Second-Order Logic	System F

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

A Logical Curiosity

$$\frac{}{\Gamma \vdash \mathsf{z} : \mathbb{N}}\ \mathbb{N}\mathsf{I}_z \qquad\qquad \frac{\Gamma \vdash e : \mathbb{N}}{\Gamma \vdash \mathsf{s}(e) : \mathbb{N}}\ \mathbb{N}\mathsf{I}_s$$

$$\frac{\Gamma \vdash e_0 : \mathbb{N} \qquad \Gamma \vdash e_1 : X \qquad \Gamma, x : X \vdash e_2 : X}{\Gamma \vdash \mathsf{iter}(e_0, \mathsf{z} \to e_1, \mathsf{s}(x) \to e_2) : X}\ \mathbb{N}\mathsf{E}$$

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

An Introduction to Agda

- 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`

Agda: Basic Datatypes

```
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

```
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

Agda: Polymorphic Datatypes

```
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

Agda: Indexed Datatypes

```
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

```
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

Agda: Indexed Datatypes

```
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: Indexed Datatypes

```
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

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

- `a ≡ b` is the type of proofs that `a` and `b` are equal
- The constructor `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

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

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

z-+-left-unit : (n : Nat) → (z + n) ≡ n
z-+-left-unit n = refl
```

- `z + n` evaluates to `n`
- So Agda considers these two terms to be identical

A Manual Theorem

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

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

z-+-right-unit : (n : Nat) → (n + z) ≡ n
z-+-right-unit z = refl
z-+-right-unit (s n) = cong s (z-+-right-unit n)
```

- We prove the right unit law inductively
- Note that *inductive proofs are recursive functions*
- To do this, we need to show that equality is a congruence

The Equality Toolkit

```
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-+-right : (n : Nat) → (n + z) ≡ n
z-+-right z = refl
z-+-right (s n) =
   cong s (z-+-right n)

s-+-right : (n m : Nat) →
            (s (n + m)) ≡ (n + (s m))
s-+-right z m = refl
s-+-right (s n) m =
  cong s (s-+-right n m)

+-comm : (i j : Nat) →
         (i + j) ≡ (j + i)
+-comm z j = z-+-right j
+-comm (s i) j = trans p2 p3
  where p1 : (i + j) ≡ (j + i)
        p1 = +-comm i j
        p2 : (s (i + j)) ≡ (s (j + i))
        p2 = cong s p1
        p3 : (s (j + i)) ≡ (j + (s i))
        p3 = s-+-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

Conclusion

- 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...