Lambda Calculus & Computability: Tip of the Iceberg

Robert M. Keller
Harvey Mudd College
11 December 2007

The fathers of computability:

- Kurt Gödel, 1931
- Alan Turing, 1936
- Stephen Kleene, 1936
- Alonzo Church, 1936
- Emil Post, 1936

Models of computability:

- Kurt Gödel, 1931: **Primitive recursive functions**
- Alan Turing, 1936: **Turing machines**
- Stephen Kleene, 1936:
  General recursive functions
  Partial recursive functions
- Alonzo Church, 1936: **Lambda calculus**
- Emil Post, 1936: **Tag systems**
- The last four models were subsequently shown to be equivalent. The first model is properly contained.

Bases for computability:

- **Symbol strings, tapes**:
  - Alan Turing, 1936: Turing machines
  - Emil Post, 1936: Tag systems
  - A.A. Markov, 1954: Formal algorithms

- **Natural Numbers**:
  - Kurt Gödel, 1931: Primitive recursive functions
  - Stephen Kleene, 1936:
    - General recursive functions
    - Partial recursive functions
  - Hermes, 1954; Minsky, 1961; Sheperdson & Sturgis, 1963:
    - Register machines

- **Anonymous Functions**:
  - Alonzo Church, 1936: Lambda calculus
  - Curry, 1956: Combinatory logic

Importance of Lambda Calculus

- Most programming languages based on it in some form.
- Thus provides a semantic link between programming languages and computability.
- It is a very “pure” model.

Lambda Calculus Line of Descent

- Lisp
- Scheme
- ML
- rex
- Haskell
- and many others

Why λ?

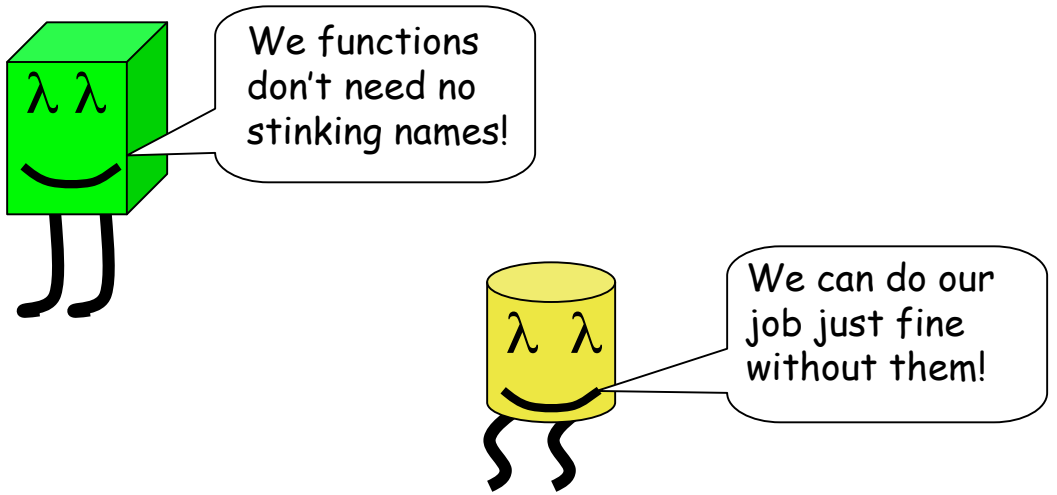
- Church wanted a carat *atop* the variable symbol x, but the printer gave him ^x, which became λx. So we are stuck with cryptic notation because of typesetters.)

- A more suggestive symbol would have been something like ⇒ to denote **function abstraction**: the function that, with argument x, gives E:

  - λ Calculus: λx.E
  - SML \x.E
  - Haskell \x E
  - Lisp, Scheme (lambda (x) E)
  - Rex: x ⇒ E (influenced by Bourbaki |→ notation)

All emphasize that **functions don't need names**.

Anonymous Functions Equity Association



Why λx.E and not just E?

- E is any expression. It is not necessarily a function.
- λx.(x + 5) is the function that adds 5 to its argument (signified by x).
- X + 5 is not a function. It is just an expression.

Oh, but its obvious …

- . . . what function we mean by x + 5.

- Well how about $x + y^c$. What function is that?
  - $F(x, y) = x + y^c$?
  - $F(y, x) = x + y^c$?
  - $F(x) = x + y^c$?
  - $F(y) = x + y^c$?
  - $F(c) = x + y^c$?
  - $F(x, y, c) = x + y^c$?

.

.

.

The λ calculus makes everything precise.

- Also, it is extremely elegant and minimal.
- But, it is also a form of "expression plumbing" in many ways.

One-argument functions suffice.

- We are accustomed to using functions with multiple arguments:

  Let $f(x, y) = x + y^5$

- The λ calculus contents itself with 1-argument functions.

One-argument functions suffice.

- Multi-argument functions are achievable by "peeling" the arguments sequentially:

  $f(x, y) = x + y^5$

  becomes

  $\lambda x.(\lambda y.(x + y^5))$

  "the function that, with argument x, yields the function that, with argument y, yields the value of $x + y^5$.

  This is called "Currying" the arguments, after Haskell B. Curry (although first introduced by Shönfinkel, 1924.)

β reduction

- In λ calculus, applying a function to an argument is called β **reduction**.
- $(\lambda x.(x + 5))(9) \rightarrow 9 + 5$ (β reduction)
- $(\lambda x.(\lambda y.(x + y^3)))(5)(9) \rightarrow (\lambda y.(5 + y^3))(9)$ (β)

  $\rightarrow 5 + 9^3$ (β)

So Far:

- λ **abstraction** (function from expression)

  λx.E from some expression E

- β **reduction** (applying a function)

  (λx.E) F → E[x/F]

  (E with ***free*** x's replaced by F)

Additional **sanity requirement (SR)**:

No variable free in F can become bound
as a result of substitution.

Free vs. Bound

- Basically the same as in predicate calculus, except λ instead of ∀ and ∃, is the binding operator.

Context-Free Grammar for λ Terms

- Let V be the set of variable symbols.
- T will derive the set of λ terms:
  - T → V (Every variable is a term.)
  - T → λV.T (Every abstraction is a term.)
  - T → T T (Every application is a term.)

  (Here → is production, *not* reduction.)
- Grouping conventions (to avoid so many parens):
  - **Abstraction groups from right-to-left**:

    λx.λy…λz.E means λx.(λy.(…λz.E)…))

  - **Application groups from left-to-right**!

    E F G means ((E F) G)

Where are the primitive functions?

- While we could add them, it turns out, in a certain sense, that we don’t need them for computability.

More examples (1)

- $\lambda x.5$ is the constant function with value 5.
- $(\lambda x.5)E \rightarrow 5$ (β), regardless of E.

- $\lambda x.E$ is the constant function with value E.
- $(\lambda x.E)F \rightarrow E$ (β), regardless of F.

- $(\lambda x.E)E \rightarrow$

More examples (2)

- $\lambda x.x$ is the identity function, since
- $(\lambda x.x)E \rightarrow E$ (β), regardless of E.
  (Note that the SR automatically holds.)

- $(\lambda x.x)(\lambda x.x) \rightarrow$

More examples (3)

- Although not required to do so, we often use upper case letters as a hint that the variable will ultimately be applied as a function:

- $\lambda F.\lambda x.Fx$
  The function that, with argument F, yields
  the function that, with argument x, yields the value of
  F applied to x.

- $(\lambda F.\lambda x.Fx)(\lambda y.y)5 \rightarrow (\lambda x.(\lambda y.y)x)5 \rightarrow (\lambda y.y)5 \rightarrow 5$
- $(\lambda F.\lambda x.Fx)(\lambda F.\lambda y.Fy) \rightarrow \lambda x.Fx[F/(\lambda F.\lambda y.Fy)] =$

$\lambda x.((\lambda F.\lambda y.Fy)x)$

Do we stop now?

Inner reductions are possible

- $\lambda x.((\lambda F.\lambda y.Fy)x)$

  There is no argument available to which to apply this function.

- However, we can do an *inner* reduction anyway:

  $(\lambda F.\lambda y.Fy)x \rightarrow \lambda y.Fy[F/x] \rightarrow \lambda y.xy$

  Giving overall:

  $\lambda x.(\lambda y.xy)$

- Note: This is something most programming languages **don't** do. They mostly work with outer reductions.

Mnemonic for the idiomatic combination: applying an abstraction

- λx.λy. . . . λz.E F . . . G H (recall the body is (((E F)…G)H)

  Peel off the **leftmost** λx. and replace free occurrences of x in the body E F . . . G with the **rightmost** term H.

- Provided, of course, no free variable in H becomes bound: the SR).

- Repeat as much as possible.

- The number of iterations is the minimum of the number of variables and the number of body terms E F . . . G H.

  - If there are fewer variables, some body terms are left.
  - If there are fewer body terms, some variables are left.

Equality of Expressions

- We will be content with an informal definition for now:

  Two expressions are considered equal (=) when they represent the same function.

- Example:

  $\lambda F.\lambda G.\lambda x.F(Gx) = \lambda G.\lambda F.\lambda x.G(Fx)$

α conversion = renaming

- In some cases, reduction can be blocked by the the SR (Sanity Requirement): This example is similar to the previous:
- $(\lambda F.\lambda x.Fx)(\lambda F.\lambda x.Fx) \rightarrow \lambda x.Fx[F/(\lambda F.\lambda x.Fx)] =$

  $\lambda x.((\lambda F.\lambda x.Fx)x)$
- Here we **cannot** do the inner reduction, because the SR prevents the substitution:

  $(\lambda F.\lambda x.Fx)x \rightarrow (\lambda x.Fx)[F/x]$ x would become bound!
- Yet in terms of **functions**, the two examples are the same.

α conversion = renaming

- We are allowed to circumvent the SR by renaming the bound variable (so long as that doesn't introduce other conflicts.):
- $\lambda x.((\lambda F.\lambda x.Fx)x) \rightarrow \lambda x.((\lambda F.\lambda y.Fy)x) \quad (\alpha)$
  $\rightarrow \lambda x.(\lambda y.xy)$
- In a sense, variables just "get in the way". There are certain formalisms that avoid them entirely.
- Often authors will α rename variables without explicit mention of it.

Advice to the Perplexed

- When possible, try to understand the expressions as **functions**, rather than in terms of reductions they might yield.
- It is easier to construct functions than it is to analyze them.
- Note that some functions will be higher-order, in that they can operate on other functions and return functions as values.

Examples

- The *compose* function: Takes two arguments (in sequence), producing the result of the composing the first with the second:

  $\lambda F.\lambda G.\lambda x.F(Gx)$

Exercises

- The twice function, which applies its first argument twice in succession to a second argument:

- What is twice applied to twice?

Church Numerals

- To establish that the is a universal computing model, Church invented a way to represent arbitrary (natural) numerals using just function composition:

  - 0 is represented as $\lambda f.\lambda x.x$
  - 1 is represented as $\lambda f.\lambda x.fx$
  - 2 is represented as $\lambda f.\lambda x.ffx$

  - n is represented as $\lambda f.\lambda x.f^n x$
    (The n-fold application of f to x.)

More Abbreviations

- These expressions are starting to get long, so we eliminate extra dots and write, e.g.

  $\lambda fx.ffx$

  instead of

  $\lambda f.\lambda x.ffx$

  again grouping abstractions to the right.

Example: The Successor Function

- To create a successor function for Church numerals, we need a function that takes
  $\lambda f.\lambda x.f^n x$ to $\lambda f.\lambda x.f^{n+1} x$
- We can derive such a function as follows:
  - Apply the left-hand expression to a function f, giving
    $\lambda x.f^n x$
  - Compose that same function f with the result above:
    compose(f, $\lambda x.f^n x$)
  - to get
    $\lambda x.f^{n+1} x$
  - So the *form* of successor (where N is the argument numeral) is:
    $\lambda N.\ \lambda f.\ \lambda x.$ (compose(f, (Nf))x)
- Recall that compose can be represented
  $\lambda F.\lambda G.\lambda x.F(Gx)$
- So successor = $\lambda N.\ \lambda f.\ \lambda x.\ (\lambda F.\lambda G.\lambda x.F(Gx))\ f\ (N\ f)\ x$
- This can be simplified by three inner reductions to:
  $\lambda N.\ \lambda f.\ \lambda x.\ f(N\ f\ x)$
- Check this:
  successor 0 = $(\lambda Nfx.f(N\ f\ x))(\lambda f.\lambda x.x)$
  $\rightarrow \lambda fx.f((\lambda f.\lambda x.x)\ f\ x)$
  $\rightarrow \lambda fx.f((\lambda x.x)x)$
  $\rightarrow \lambda fx.f(x) = 1$

Church Addition

- Addition of two numbers is applying a particular function first to the (Church representation for) the first number, then that result to (the Church representation for) the second:

  $(\lambda f.\lambda x.f^m x) + (\lambda f.\lambda x.f^n x) \rightarrow (\lambda f.\lambda x.f^{m+n} x)$

- Try a strategy similar to that for successor:

  add(M, N) = λf.compose(Mf, Nf)

- So

  add= λM. λN. λf. λx.(M f)((N f)x)

  or simply add= λMNfx. M f (N f x)

- Exercise:

  Devise an expression that multiplies Church numerals

Church Booleans

- Just as numbers can be represented by functions, so can truth values.
- Equate:

  T = λxy.x (return first of two arguments)

  F = λxy.y (return second of two arguments)

  Then:

  and = λxy.xy(λuv.v)

  or = λxy.x(λuv.u)y

  not = λx.x(λuv.v)(λab.a)

Church zero-tester

- $Z = \lambda x.x\ (F\ not)\ F$

  $F = \lambda xy.y \qquad not = \lambda x.x(\lambda uv.v)(\lambda ab.a)$

- It can be worked out that:

  $Z\ zero = T$

  $Z\ n = F$ for any non-zero n

Can the λ calculus compute any computable function?

- We have a start.
- But in general, we'll need arbitrary iteration.
- How to get it?

Can the λ calculus compute any computable function?

- Iteration can be done with recursion, which will be needed anyway.
- How do we get recursion?
- There is no obvious way to define a function using its own definition.
- After all, functions are anonymous, so there is no obvious way for it to refer to itself.

Self-Reference through the back-door

- Although a function can't refer to itself by name, if a variable is bound to the function, the function could apply to the same variable, hence to itself.
- Example:

  $\lambda F.FF$

  applies its argument to itself.

Self-Reference through the back-door

- Seemingly useless example:

  $(\lambda F.FF)\ (\lambda F.FF) \rightarrow$

- What if the argument to $(\lambda F.FF)$ is a **functional**:

  $(\lambda G.\lambda x.E)$

  $(\lambda F.FF)\ (\lambda G.\lambda x.E)$

  $\rightarrow (\lambda G.\lambda x.E)\ (\lambda G.\lambda x.E)$

  $\rightarrow \lambda x.E[G/(\lambda G.\lambda x.E)]$

  So in the appropriate context, this functional has access to **itself**, in the form of occurrences of variable G in E.

Example from Programming

- fac(n) = n < 1 ? 1 : n*fac(n-1)
- fac *appears* to require self-reference.
- First re-cast fac using a Curried functional G:

  G(F)(n) = n < 1 ? 1 : n*F(n-1) (So G(fac) = fac)

- If we can supply G(F) as the argument F, we will get the ***effect*** of recursion: (n < 1 ? 1 : n*F(n-1))[F/G(F)]
  = (n < 1 ? 1 : n*(n-1 < 1 ? n*F(n-1-1))) [F/G(F)] =
  (n < 1 ? 1 : n*(n-1 < 1 ? n*(n-1-1 < 1 ? 1 : …))) [F/G(F)]

Example from Programming

- Re-cast fac using a Curried functional E:

  G(F)(n) = n < 1 ? 1 : n*F(n-1)

- If we can supply G(F) as the argument F, we will get the effect of recursion.
- i.e. we need G applied to G(F), with F as G applied to G(F), with … .
- In short, we need a function of G that gives the **effect** of G(G(G(…)))
- Call this function Y, as if Y(G) = G(Y(G)),
- For then Y(G) = G(Y(G)) = G(G(Y(G)) = G(G(G(Y(G)))) = …
  as desired.

Fixed Point Operators

- A function Y with the property that, for **any** G:

  $$Y(G) = G(Y(G))$$

  is called a fixed-point operator.

  There is an infinite set of them in the lambda calculus.

  The canonic example is: $Y = \lambda\, g.\ (\lambda\, x.\ g\ (x\ x))\ (\lambda\, x.\ g\ (x\ x))$

  $Y(G) \rightarrow (\lambda\, x.\ G\ (x\ x))\ (\lambda\, x.\ G\ (x\ x))$

  $\rightarrow G\ ((\lambda\, x.\ G\ (x\ x))\ (\lambda\, x.\ G\ (x\ x)))) = G(Y(G))$

  This Y is sometimes called the “paradoxical combinator”.

Factorial without recursion

- The factorial functional is

  G(F)(n) = n < 1 ? 1 : n*F(n-1)

- The fixpoint operator has the property

  Y(G) = G(Y(G))

- So letting F = Y(G) = G(Y(G)), F(n) should be factorial(n) for the G above.
- On the left-hand side:

  G(F)(n) = G(Y(G))(n) = F(n)

- So we have the desired equality:

  F(n) = n < 1 ? 1 : n*F(n-1)

Test our theory using rex

```
// Showing implementation of fixed points for functionals in rex

// The factorial functional:

G(F) = ((n) => ((n < 1) ? 1 : (n*F(n-1))));

// The functional is meaningful on its own:

test(G(id)(0), 1);
test(G(G(id))(1), 1);
test(G(G(G(id)))(2), 2);
test(G(G(G(G(id))))(3), 6);
test(G(G(G(G(G(id)))))(4), 24);
test(G(G(G(G(G(G(id))))))(5), 120);

// With sufficiently-many nested G's, we'll always get the value
// of the factorial. The problem is that no finite nest works
// for all arguments.

// A fixed-point operator Y1 (which uses recursion in its definition).
// The $ defers evaluation of the argument until it is needed.

Y1(G) = G($Y1(G));

test(Y1(G)(5), 120);


// The canonic lambda calculus Y operator
// The $ defers evaluation of the argument until it is needed.

Y2(g) = (((x) => g($x(x)))((x) => g($x(x))));

test(Y2(G)(5), 120);
```

Execution:

```
knuth > ~keller/pub/rex lambda.rex
ok: (G id)(0) ==> 1
ok: (G (G id))(1) ==> 1
ok: (G (G (G id)))(2) ==> 2
ok: (G (G (G (G id))))(3) ==> 6
ok: (G (G (G (G (G id)))))(4) ==> 24
ok: (G (G (G (G (G (G id))))))(5) ==> 120
ok: (Y1 G)(5) ==> 120
ok: (Y2 G)(5) ==> 120
lambda.rex loaded
```

Two key insights were needed

- Recursive definitions can be achieved as fixed points of functionals (functions that take functions as arguments).

- Fixed-point operators can be constructed in λ calculus.

Partial recursive functions

- This family of partial functions on the natural numbers can be shown to be equipotent to Turing machines (each can simulate the other). Also:

- **Every partial-recursive function can be represented in the λ calculus using Church numerals to represent numbers.**

- We thus have **Church's thesis**: The λ calculus can implement any computable function.

Church vs. Turing Thesis

- Church's version is not as compelling as Turing's.
- Thus Church's might more rightly be called a "hypothesis" than an assertion.
- In particular, Gödel found Turing's argument more compelling.
- Luckily for Church the two models are in some sense equivalent.

Partial recursive functions

- A related family, also equipotent to the others, is Kleene's "**general recursive functions**". These are similar to definition of numeric functions in typical functional languages.

- The partial recursive functions (also defined by Kleene) are more rigidly structured.

- They are based off of the **primitive recursive functions** (as they are now called) originally due to Gödel and used in his proof of the incompleteness theorem.

Statement of the result

- The definition of partial recursive function follows, but since the presentation is long, we'll state the result here.
- Also, we don't give the proof of the result in these slides. However, most of the groundwork has been laid in the preceding discussion of the λ calculus.

Primitive Recursive Functions

- The set of primitive recursive functions is given inductively.
- Every function is k-ary, for some $k \geq 0$.
- The domain and co-domain of each function is the set of natural numbers {0, 1, 2, 3, …}.

Basis Functions (1 of 3)

- The **zero** functions are all primitive recursive:

$$Z^k(x_1, x_2, \ldots x_k) = 0$$

for each arity $k \geq 0$.

Basis Functions (2 of 3)

- The **projection** functions are all primitive recursive:

$$P^k_j(x_1, x_2, \ldots x_k) = x_j$$

for each arity $k \geq 0$ and each i, $1 \leq i \leq k$.

Basis Functions (3 of 3)

- The **successor** function is primitive recursive:

$$S(x) = x + 1$$

Induction Rules (1 of 2)

- The **composition** of primitive recursive functions is primitive recursive:

$$h(x_1, x_2, \ldots x_k) =$$

$$f(g_1(x_1, x_2, \ldots x_k),$$
$$g_2(x_1, x_2, \ldots x_k),$$
$$\ldots$$
$$g_r(x_1, x_2, \ldots x_k))$$

for each pair of arities k, r $\geq$ 0.

Constant Functions

- A consequence of the rules up to this point is that **constant** functions are all primitive recursive:

$$C^k_c(x_1, x_2, \ldots x_k) = c$$

for each natural number c.

This is so because is just a composition of the zero and successor functions:

$C^k_c(x_1, x_2, \ldots x_k) = S(S(\ldots S(zero(x_1, x_2, \ldots x_k)) \ldots))$

Explicit Definition (ED)

- This is a plumbing convenience. In lieu of showing nests of compositions, projections, and constants, we can just use definitions such as:

  $f(x, y, z) = g(h(y, x), 5, k(z, z))$

  and know that if g, h, and k are primitive recursive, so is f.

  ED is also called “Explicit Transformation” (ET).

Induction Rules (2 of 2)

- A function defined from primitive recursive functions by the following **primitive recursion pattern** is primitive recursive :

$$h(0, x_1, x_2, \ldots x_k) = g(x_1, x_2, \ldots x_k)$$

$$h(n+1, x_1, x_2, \ldots x_k) =$$

$$f(x_1, x_2, \ldots x_k, n, h(n, x_1, x_2, \ldots x_k))$$

- Here h is being defined from g and h, which are known to be primitive recursive.

Examples of Primitive Recursive Functions

- add(x, y): addition
- mult(x, y): multiplication
- pred(x): predecessor
- sub(x, y): proper subtraction
- mod(x, y): modulus
- div(x, y): integer division (quotient)
- sqrt(x): integer square root

rex implementations

- I will demonstrate some of these using rex rules. This allows the definitions to be tested readily.
- rex does not enforce the primitive recursive formalism
- We have to be careful not to “cheat”.

add implementation

- S(n) = n + 1; // pretend this is built in
- add(0, y) => y;
- add(n+1, y) => S(add(n, y));
- For reference, the primitive-recursive pattern is:

$$h(0, x_1, x_2, \ldots x_k) = g(x_1, x_2, \ldots x_k)$$

$$h(n+1, x_1, x_2, \ldots x_k) =$$

$$f(x_1, x_2, \ldots x_k, n, h(n, x_1, x_2, \ldots x_k))$$

mult implementation

- mult(0, y) => 0;
- mult(n+1, y) => add(y, mult(n, y));
- For reference

$$h(0, x_1, x_2, \ldots x_k) = g(x_1, x_2, \ldots x_k)$$

$$h(n+1, x_1, x_2, \ldots x_k) =$$

$$f(x_1, x_2, \ldots x_k, n, h(n, x_1, x_2, \ldots x_k))$$

pred implementation

- pred(0, y) =>
- pred(n+1, y) =>
- For reference

$$h(0, x_1, x_2, \ldots x_k) = g(x_1, x_2, \ldots x_k)$$

$$h(n+1, x_1, x_2, \ldots x_k) =$$

$$f(x_1, x_2, \ldots x_k, n, h(n, x_1, x_2, \ldots x_k))$$

sub implementation

- sub is **proper** subtraction (aka “monus”):
  If a < b, then sub(a, b) = 0.
- sub(y, 0) =>
- sub(y, n+1) =>
- For reference

$$h(0, x_1, x_2, \ldots x_k) = g(x_1, x_2, \ldots x_k)$$

$$h(n+1, x_1, x_2, \ldots x_k) =$$

$$f(x_1, x_2, \ldots x_k, n, h(n, x_1, x_2, \ldots x_k))$$

Primitive Recursive Predicates

- For some definitions we want to have predicates, which we can equate to functions return only values 0 (false) and 1 true.
- sgn(0) => 0;
- sgn(n+1) => 1;
- sgn converts arbitrary values to {0, 1}.

Negation

- not(0) => 1;
- not(n+1) => 0;

Equality Predicate

- eq(x, y) = not(add(sub(x, y), sub(y, x)));

if-then-else function

- ifthenelse(0, x, y) => y;
- ifthenelse(n+1, x, y) => x;

- This can be inefficient if all arguments must be computed to get a result (applicative order). It should be taken as an “academic” version, perhaps.

mod and div

- mod(0, y) => 0;
- mod(n+1, y) => ifthenelse(eq(S(mod(n, y)), y),
  0,
  S(mod(n, y)));
- div(0, y) => 0;
- div(n+1, y) => ifthenelse(eq(S(mod(n, y)), y),
  S(div(n, y)),
  div(n, y));

Perspective

- Primitive recursive functions are functions that can be defined using only **definite iteration** (e.g. for-loop with upper bound pre-determined)

  and not requiring indefinite iteration (while-loops) or the full power of recursion.

- Primitive recursion as given is not a special case of tail recursion, although there is an equivalent version that is.

Primitive Recursion = Definite Iteration

- The function h defined in the primitive recursion scheme can be computed by the following for-loop:

$$\text{acc} = g(x_1, x_2, \ldots x_k);$$
$$\text{for}(j = 0; j < n; j{+}{+})$$
$$\qquad \text{acc} = f(x_1, x_2, \ldots x_k, j, \text{acc});$$

$$// \text{ at end: acc} == h(n, x_1, x_2, \ldots x_k)$$

Tail-Recursive Version

- The function h(n, x_1, x_2, . . . x_k) can be computed as t(n, g(x_1, x_2, . . . x_k)) where t is defined in the following tail-recursion:

  t(0, acc) = acc;

  t(n+1, acc) = f(x_1, x_2, . . . x_k, n, acc);

Example: Factorial

- Primitive-recursive version
  (uses the primitive-recursion pattern):

  fac(0) = 1
  fac(n+1) = mult(n+1, fac(n))

- Tail-recursive version
  (doesn't use the pattern, but equivalent):

  fac_tr(n) = t(n, 1)

  t(0, acc) = acc
  t(n+1, acc) = t(n, mult(n+1, acc))

Tail-Recursion Theorem

- If h is defined from f and g using primitive recursion, then h can also be defined from f and g using tail recursion.
- The conversion is given on preceding slides.
- Claim: $(\forall n)\ t(n, acc) = h(n, x_1, x_2, \ldots x_k)$ provided acc is initialized with $g(x_1, x_2, \ldots x_k)$.
- Proof is by induction on n.
  Show that both

  $t(n, acc) = f(X, n\text{-}1, f(X, n\text{-}2, \ldots, f(X, 0, g(X))\ldots))$

  $h(n, X) = f(X, n\text{-}1, f(X, n\text{-}2, \ldots, f(X, 0, g(X))\ldots))$

Totality Theorem

- Every primitive recursive function is total.
- Two levels of induction are involved:
  - For each use of the primitive-recursion pattern, there is an induction to show that h is defined for all n, assuming that f and g are total.
  - There is induction on the number of uses of the induction rules in defining the top level function.

Computability Theorem

- Every primitive-recursive function is computable by a Turing machine.

- This is more-or-less obvious, but it can be shown in significant detail by showing how a Turing machine can be constructed by composing functions using the basis functions and induction rules.

Diagonalization Theorem

- There is a total recursive function that is not primitive recursive.
- Proof: Using the computability theorem, we know that we can enumerate the primitive recursive functions of one argument:

  $p_0, p_1, p_2, \ldots$

  They are just a subsequence of the ones computed by Turing machines in the ordering of all Turing machines.

  Then define $q(x) = p_x(x) + 1$. This function is clearly total, since each p_x is, but q cannot appear in the list.

The Ackermann Hierarchy

- We notice that add and mult have similar definitions.
  - add uses S as a base
  - mult uses add as a base
- We can go on to define exp analogously, using mult as a base.
- When does this stop?
- We quickly reach functions that have very large values for small arguments.
- It is possible to diagonalize over this hierarchy.

The Ackermann Hierarchy

- $A_0(m) = S(m)$
- $A_{n+1}(0) = A_n(1)$
- $A_{n+1}(m+1) = A_n(A_{n+1}(m))$

- Each function in the list: A_0, A_1, A_2, . . . is clearly primitive-recursive.
- Define $A(n, m) = A_n(m)$.
- It can be proved that A **grows faster** than any single primitive-recursive function, hence is not primitive-recursive itself.

μ Recursive Functions

- This is a family of partial functions, also known as "**the** partial recursive functions".

- We have used that term to describe the partial computable functions, and the definitions turn out to be equivalent.

μ-Recursive Functions

- Start with the primitive-recursive functions as a base.
- Add one more induction rule: If f is a k+1 ary μ–recursive function, h is a a k+1 ary one:

$h(x_1, x_2, \ldots x_{k-1}) =$

$$\mu x_k \, [f(x_1, x_2, \ldots x_k) = 0]$$

read "the least value of x_k such that $f(x_1, x_2, \ldots x_k) = 0$".

μ-Recursive Functions

- The definition of a function using the operator really only makes sense if the function f is **total**.
- Totality cannot be established syntactically (why?).
- We will adopt the convention that the values of for which f is computed are given sequentially, and that if $f(x_1, x_2, \ldots x_k)$ is divergent for any value before reaching a value x_k such that $f(x_1, x_2, \ldots x_k) = 0$, then $h(x_1, x_2, \ldots, x_{k-1})$ also diverges for the given arguments.

Computability Theorem for μ-Recursive Functions

- We can extend to the partial recursive functions the proof that the primitive recursive functions are Turing computable.

Converse of the Computability Theorem

- Every Turing computable partial function is computable by a μ-recursive partial function.

- Moreover, the μ operator needs to be used only **once** to achieve any partial-recursive function.

Importance of the Computability Therem and its Converse

- Turing-computable partial functions and μ-recursive partial functions are established as being the same thing.

- One was defined using **strings**, the other using **numbers**.

Establishing the Converse

- The converse shows that any Turing-computable partial function is a μ-recursive partial function.
- To do this involves **encoding** TM tapes and configurations as numbers.
- Then it can be shown that there are primitive recursive functions that:
  - Simulate a single step of a Turing machine.
  - Tell whether an encoded configuration is halting.

Primitive Recursive TM equivalents

- R(x) is the configuration resulting after 1 step from x.
- T(i, x) is the configuration resulting from configuration x after i steps.
- P(x) indicates whether or not a configuration is halting (0 or 1).

Recursive TM equivalents

- Halting in i steps is expressed by:

$$\mu i\,[P(T(i, x_0)) = 0]$$

- The halting configuration is:

$$T(\mu i\,[P(T(i, x_0)) = 0], x_0)$$

Encodings

- Using **primitive** recursive functions to encode and decode tapes and configurations requires a lengthy, but interesting, excursion.

- One way (but not the only way) to encode arbitrary sequences of numbers is to use "Gödel numbering":

  Any sequence of natural numbers

  $$(x_1, x_2, \ldots, x_k)$$

  can be encoded as a **single** natural number:

  $$p_1^{x1} p_2^{x2} \ldots p_k^{xk}$$

Universal μ-Recursive Functions

- Most results for Turing machines have parallels for the μ-Recursive Functions.

- The μ-recursive functions are programs that can be coded and enumerated just like Turing machines can:

  $\varphi^k_0, \varphi^k_1, \varphi^k_2, \varphi^k_3, \ldots$

  are the k-ary μ-recursive functions for any fixed k.

Kleene's Normal Form Theorem

- For each $k \geq 1$, there exists a 1-ary primitive recursive function U and a (k+2)-ary primitive recursive predicate T_k such that
  - $\varphi^k_n(x_1, x_2, \ldots, x_k)\downarrow$ iff $(\exists z)\ T(n, x_1, x_2, \ldots, x_k, z)$

    $\varphi^k_n(x_1, x_2, \ldots, x_k) = U(\mu z\ [T(n, x_1, x_2, \ldots, x_k, z) = 0])$
- Essentially, T is like the function that tells whether the n^{th} configuration of a TM computation is halting, while U gives the result from that halting configuration.
- The numbers z code both the program **and** the number of steps.

Universal μ-Recursive Functions

- For each k, there is a μ-recursive function ψ of k+1 variables such that

$$\psi(n, x_1, x_2, \ldots, x_k) =$$

$$\varphi^k{}_n(x_1, x_2, \ldots, x_k)$$

- ψ is a universal function for k arguments.

Recursive and R.E. Sets

- Languages are now sets of numbers.
- A set is recursive if its characteristic function is total recursive.
- A set is recursively-enumerable if there is a total recursive function that enumerates it.
- Equivalently, a set is recursively-enumerable if it is the domain of some partial recursive function.

Halting and Divergence

- $\varphi(x)\downarrow$ is used to mean that φ has a value for argument x.
- $\varphi(x)\uparrow$ is used to mean that φ diverges on argument x.
- The set $\{j \mid \varphi_j(j)\uparrow\}$ is not recursively-enumerable: this is the **halting problem**.
- The set $\{j \mid \varphi_j(j)\downarrow\}$ is recursively-enumerable, but not recursive.
- The set $\{j \mid \varphi_j(0)\uparrow\}$ is not R.E. either, analogous to the blank-tape halting problem.

Problem: Exhibit a total function that is not computable.

- Define

$$f(j) = \begin{cases} \varphi_j(j) + 1 \text{ if } \varphi_j(j)\downarrow \\ \\ 0 \text{ otherwise} \end{cases}$$

f is evidently total. f is not computable, since if it were, say φ_k , then f(k) would give a contradiction.

Hilbert's Tenth Problem

- This is an unsolvable problem of practical interest.
  - Give an algorithm that will determine whether a multivariate polynomial equation has an **integer** solution.
  - Example of an equation:
    $x^2 + 3y^3 + 13 = 0$
- The problem of whether this is possible was posed by Hilbert in 1900.
- This problem was not proved unsolvable until 1970, when it was established that **every** recursively-enumerable set of k-tuples is the set of solutions of some such equation.