CS655 Lecture 12 λ Calculus (Part 2) and Introduction to Type Systems 2 March, 2006

Scribe: Joseph Carnahan Lecturer: Westley Weimer

1 Class Announcements

In case anyone needs a review, the 3 (or 4) things required for this class are

- taking scribe notes,
- the homework assignments,
- the semester project, and
- a (possible) final exam.

Also, as you could probably infer, the fact that some of the readings are marked as "optional" is meant to imply that the other readings are not optional.

2 Outline of Today's Lecture

- λ-calculus
  - Applications
  - Encodings
  - (Another) Use of fixed points
- Type systems
  - Static vs. dynamic
  - Safety, judgements, etc.

Quote for the day: "The purpose of a type system is to reject bad programs."

3 λ-Calculus Review

The λ-calculus centers around functions and their evaluation. The grammar is simple:

$$\begin{array}{rcll} e & ::= & x & \text{(variable)} \\ & & \mid \lambda x.\, e & \text{(definition of a function with parameter } x\text{)} \\ & & \mid e_1\, e_2 & \text{(function } e_1 \text{ applied to an argument } e_2\text{)} \end{array}$$

λ-calculus expressions are evaluated by a procedure called *β-reduction*, which applies capture-avoiding substitution to replace bound variables in functions by the values that have been given for them as the functions' arguments. *Capture-avoiding* means that scoping rules are observed: If a function f_1 has a parameter x and contains another function f_2 with a parameter named x, the two x's are considered distinct. This means that when f_1 is called with an argument y, only the x's outside of the body of f_2 are replaced with y. (This kind of substitution can be made simpler by renaming variables so that no two functions have a formal parameter with the same name.)

Note that the λ-calculus does not include integers, booleans, if or while statements, or many other commonly-used programming constructs. Later, we will look at how these constructs can be created using only the tools that the λ-calculus provides.

4 Functional Programming

In *functional programming*, functions are first-class objects. They can be passed as arguments to other functions, and they can also be returned as results. Functional programming is considered a "higher-order" way to program: Instead of just writing functions to manipulate data, you can write functions to manipulate other functions.

Functional programming is used to different degrees in different languages, going from functional languages that have added some imperative features (Lisp, Scheme, ML) to imperative languages that have added some functional features (Python, Ruby). "Pure" functional programming would be programming directly in the λ-calculus, where functions are the *only* kind of data that functions can manipulate.

One property that measures the purity of a functional programming language is *referential transparency*. Referential transparency means that you can evaluate all expressions and definitions of variables using (capture-avoiding) substitution:

$$(\mathsf{let}\ x = e_1\ \mathsf{in}\ e_2) \equiv ([e_1/x]e_2)$$

This only holds if expressions have no side-effects. For example, given an expression with a side effect like $e_1 = y{+}{+}$, this does not hold, because evaluating e_1 once only increments y once, while substituting $y{+}{+}$ for every occurence of x in e_2 might mean incrementing y any number of times in e_2.

Question: Is there a difference in bug density (bugs per line of code) between imperative and functional languages?

Answer: This has been investigated, but it's hard to quantify because different languages tend to pack more (ML, Perl) or less (Python) code onto each line.

5 Encodings Using the λ-Calculus

In spite of its humble appearances, the λ-calculus is Turing-complete, capable of encoding an arbitrary Turing machine. This encoding usually involves the trick of designing an object that computes something analogous to what a static structure would represent. For example, true is encoded as a function that discards its second argument and false is encoded as a function that discards its first argument:

$$\begin{aligned} \mathsf{true} &= \lambda x.\, \lambda y.\, x \\ \mathsf{false} &= \lambda x.\, \lambda y.\, y \end{aligned}$$

This reflects if statements where true implies taking the then branch (the first argument) and false implies taking the else branch.

5.1 Pairs and Lists

To start building data structures, we start with pairs. A pair is a function that, given a Boolean, returns its first argument for true and its second for false. From pairs, lists are constructed recursively by making a pair the second element of another pair as many times as needed to build the list. In terms of functions, we need three of them to do this, one for creating a pair, one for extracting the first half of the pair and one for extracting the second half of the pair:

$$\begin{aligned} \mathsf{mkpair}\ x\, y &\equiv \lambda b.\, b\, x\, y \\ \mathsf{fst}\ p &\equiv p\ \mathsf{true} \\ \mathsf{snd}\ p &\equiv p\ \mathsf{false} \end{aligned}$$

5.2 Arithmetic in λ-Calculus

Natural numbers are implemented as iterators. In other words, given a function f and an argument s, the number n is represented by a function that applies the function f to s n times. This representation of numbers is called Church numerals, named for Alonzo Church.

The successor function (succ) takes a number n as an argument, picks out the function f from it, and applies f to the result of evaluating n, giving you a new function that applies f a total of $n + 1$ times.

Other arithmetic operations can be constructed in this way:

- Addition is implemented by applying the successor function n_1 times to n_2.
- Multiplication is implemented by applying the addition function n_1 times to n_2.
- Exponentiation is implemented by applying the multiplication function n_1 times to n_2.

With each of these, the only tricky part is identifying the base case, the number that should be succeeded, added or multiplied first. For addition and multiplication, that number is zero, and for exponentiation, that number is 1.

5.3 Recursion

A *predicate* is a function of one argument that returns a boolean value. Now, consider a function find, which takes a predicate P and a number n and returns the smallest natural number larger than n such that P is true. Using find, we can implement while b do c by feeding the negation of the loop guard b to find and getting a number. (That number is the number of times that the loop will execute, as it's the number of times the loop would have had to have been executed for b to become false.) Then, we can pass the body of the loop to this number, which runs the body of the loop the correct number of times.

How do we implmement find? Just like other incarnations of the while loop, find satisfies the unwinding equation:

$$\mathsf{find}\ p\, n = \mathsf{if}\ p\, n\ \mathsf{then}\ n\ \mathsf{else}\ \mathsf{find}\ p\, (\mathsf{succ}\ n)$$

To evaluate this, we define a function F:

$$F = \lambda f.\, \lambda p.\, \lambda n.\, (p\, n)\, n\, (f\, p\, (\mathsf{succ}\ n))$$

The value of find is the fixed point of F, such that $\mathsf{find}\ p\, n = F\ \mathsf{find}\ p\, n$. To do this in λ-calculus, we use a special equation called the Y-combinator:

$$Y = \lambda F.\, (\lambda y.\, F\, (y\, y))\, (\lambda x.\, F\, (x\, x))$$

We can then verify that $Y\, F =_\beta F\, (Y\, F)$. In fact, the Y-combinator lets us compute a fixed-point of any λ-function, assuming that the fixed-point exists. (If the fixed-point doesn't exist, the Y-combinator will loop forever. Clearly, the Halting Problem still haunts us...)

6 Types: Making the λ-Calculus Useful

To program in the λ-calculus, we do not want to have to re-derive every other kind of data in the way we derived the boolean values and the natural numbers. To save us that work, we will introduce the following:

- constants, such as natural numbers
- types

A *type* is a summary of the the values that a variable could legally take on during the execution of a program. For example, the boolean type limits the range of legal values to true and false, and the unsigned integer type limits the range of legal values to $\mathcal{N}$.

Pure λ-calculus is untyped - Arbitrary values can be passed as arguments, and bad things may happen if an unexpected value is given to a function. At the other extreme, statically-typed languages specify what the type of each and every variable is at compile time.

The purpose of types is to prevent certain types of run-time execution errors. Run-time execution errors include two kinds:

trapped errors – Errors that halt the program

untrapped errors – Errors that do not halt the program but can instead lead to undefined behavior

Trapped errors are in some sense "well-defined", as we know that either they will halt the program or be handled by hardware and/or a signal handler. So, a program is considered *safe* if it does not cause untrapped errors.

Modern type systems let us prevent many kinds of errors:

- race conditions
- resource leaks
- insecure information flow
- etc.

Checking for these errors before run-time (*static checking*) is preferable to detecting errors in testing. However, detecting certain errors statically is undecidable. Hence, in practice most type-checkers miss some errors and detect other things as errors on paths that never will be executed. Still, languages like ML, Modula-3, and Java try to do as much static type-checking as possible.

For other errors, they can be caught with *dynamic checking*, such as compiler-inserted array bounds checks and run-time type information in C++ and Java.

Table of type-checking in different languages:

	Statically Typed	*Dynamically Typed*	*Untyped*
Safe	ML, Java, Ada, C#, Haskell, ...	Lisp, Scheme, Ruby, Perl, Smalltalk, PHP, Python, ...	λ-calculus
Unsafe	C, C++, Pascal, ...	(none)	Assembly

Note that *all* safe languages use typing in one form or another.

Today, the big questions in type research no longer center around the relative merits of static versus dynamic typing but rather with the properties that can be proven by a given type system.