Last time we introduced the λ-*calculus*, a mathematical system for studying the interaction of *functional abstraction* and *functional application.*

We discussed the syntax and some conventions regarding parsing and gave several examples. Now we arrive at the question: How does one *evaluate* a λ-calculus term? This is analogous to running a program in a functional language.

The traditional evaluation mechanism of the λ-calculus is based on the notion of *substitution.* The main computational rule is called β-*reduction.* This rule applies whenever there is a subterm of the form $(\lambda x.\, e_1)\, e_2$ representing the application of a function $\lambda x.\, e_1$ to an argument e_2. The β-reduction rule substitutes e_2 for the variable x in the body of e_1, then recursively evaluates the resulting expression.

We must be very careful about the formal definitions, however, because trouble can arise if we just substitute terms for variables blindly.

1 Scope, Bound and Free Occurrences, Closed Terms

The *scope* of the abstraction operator λx shown in the term $\lambda x.\, e$ is its *body* e. An occurrence of a variable y in a term is said to be *bound* in that term if it occurs in the scope of an abstraction operator λy (with the same variable y); otherwise, it is *free.* A bound occurrence of y is *bound to* the abstraction operator λy with the smallest scope in which it occurs.

Note that a variable can have both bound and free occurrences in the same term, and can have bound occurrences that are bound to different abstraction operators.

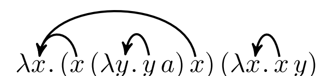

Figure 1: Scope and bindings

For example, in the term shown in Fig. 1, all three occurrences of x are bound. The first two are bound to the first λx, and the last is bound to the second λx. The first occurrence of y is bound, the a is free, and the last y is free, since it is not in the scope of any λy.

This scoping discipline is called *lexical* or *static scoping.* It is called so because the variable's scope is defined by the text of the program, and it is possible to determine its scope before the program runs by inspecting the program text.

1.1 Free Variables

Formally, the set of *free variables* of a term, denoted $FV(e)$, is defined inductively as follows:

$$FV(x) \triangleq \{x\} \qquad FV(e_0\, e_1) \triangleq FV(e_0) \cup FV(e_1) \qquad FV(\lambda x.\, e) \triangleq FV(e) - \{x\}.$$

This definition is inductive on the structure of e. The basis is the leftmost equation, and the other two are the inductive cases. In each of the two inductive cases, the right-hand side defines the value of $FV(e)$ in terms of proper subterms of e, which are smaller. Since all terms have finite size, this means that the definition eventually reaches the base case of variables. This is an example of *structural induction.* We will see many more definitions by structural induction in this course.

A term is *closed* if it contains no free variables; thus all occurrences of any variable x occur in the scope of a binding operator λx. A term is *open* if it is not closed.

2 Substitution and β-Reduction

2.1 Variable Capture

Intuitively, to perform β-reduction on the term $(\lambda x.\, e_1)\, e_2$, we substitute the argument e_2 for all free occurrences of the formal parameter x in the body e_1, then evaluate the resulting expression (which may involve further such steps).

However, we cannot just substitute e_2 blindly for x in e_1 because of the problem of *variable capture*. This would occur if e_2 contained a free occurrence of a variable y, and there were a free occurrence of x in the scope of a λy in e_1. In that case, the free occurrence of y in e_2 would be "captured" by that λy and would end up bound to it after the substitution, which would incorrectly alter the semantics.

For example, consider the substitution of x for y in $\lambda x.\, xy$. Raw substitution would yield $\lambda x.\, xx$. The variable x has been *captured* by the binding operator λx.

To prevent this, we can rename the bound variable x to z to obtain $\lambda z.\, zy$ before doing the substitution. This transformation does not change the semantics. Now substituting x for y yields $\lambda z.\, zx$; the variable has not been captured.

2.2 Safe Substitution

This idea leads to the following formal definition of *safe substitution*. The definition is by structural induction. We write $e_1\,\{e_2/x\}$ to denote the result of substituting e_2 for all free occurrences of x in e_1 according to the following rules.[1]

$$
\begin{array}{rcll}
x\,\{e/x\} & \triangleq & e & \\
y\,\{e/x\} & \triangleq & y & \text{where } y \neq x \\
(e_1\, e_2)\,\{e/x\} & \triangleq & (e_1\,\{e/x\})\,(e_2\,\{e/x\}) & \\
(\lambda x.\, e_0)\,\{e/x\} & \triangleq & \lambda x.\, e_0 & \\
(\lambda y.\, e_0)\,\{e/x\} & \triangleq & \lambda y.\, (e_0\,\{e/x\}) & \text{where } y \neq x \text{ and } y \notin FV(e) \\
(\lambda y.\, e_0)\,\{e/x\} & \triangleq & \lambda z.\, (e_0\,\{z/y\}\,\{e/x\}) & \text{where } y \neq x,\ z \neq x,\ z \notin FV(e_0), \text{ and } z \notin FV(e).
\end{array}
$$

Note that the rules are applied inductively. That is, the result of a substitution in a compound term is defined in terms of substitutions on its subterms.

The last of the six rules applies when $y \in FV(e)$. In this case, we rename the bound variable y to z to avoid capture of the free occurrence of y. One might well ask: but what if y occurs free in the scope of a λz in e_0? Wouldn't the z then be captured? The answer is that it will be taken care of in the same way, but inductively on a smaller term.

Despite the importance of substitution, it was not until the mid-1950's that a completely satisfactory definition of substitution was given by Haskell Curry. Previous mathematicians, from Newton to Hilbert to Church, worked with incomplete or incorrect definitions. It is the last of the rules above that is the hardest to get right, because it is easy to forget one of the three restrictions on the choice of z or to falsely convince oneself that they are not needed.

Rewriting $(\lambda x.\, e_1)\, e_2$ to $e_1\,\{e_2/x\}$ is the basic computational step of the λ-calculus and is called *β-reduction*. In the pure λ-calculus, we can start with a λ-term and perform β-reductions on subterms in any order.

[1] There is no standard notation for substitution. Pierce [14] writes $[x \mapsto e_2]e_1$. Other notations for the same idea are encountered frequently, including $e_1[x \mapsto e_2]$, $e_1[x \leftarrow e_2]$, $e_1[x/e_2]$, $e_1[e_2/x]$, and $e_1[x := e_2]$. Because we will be using brackets for other purposes, we will use the notation $e_1\,\{e_2/x\}$.

2.3 Safe Substitution in Mathematics

The problem of variable capture arises in many other mathematical contexts. It can arise anywhere there is a notion of variable binding and substitution.

For example, in the integral calculus, the integral operator is a binder. In the following naive attempt to evaluate a definite integral, a variable is incorrectly captured:

$$\mathsf{int}_0^x(1+\mathsf{int}_0^1 x\,dx)\,dy \;=\; (y+\mathsf{int}_0^1 yx\,dx)\Big|_{y=0}^{y=x} \;=\; (x+\mathsf{int}_0^1 x^2\,dx)-0 \;=\; x+\tfrac{1}{3}x^3\Big|_{x=0}^{x=1} \;=\; x+\tfrac{1}{3}$$

This is incorrect. The substitution of x for y under the integral in the second step is erroneous, because x is the variable of integration and is bound by the integral operator, whereas y is free. To fix this, we need only change the variable of integration to z.

$$\mathsf{int}_0^x(1+\mathsf{int}_0^1 z\,dz)\,dy \;=\; (y+\mathsf{int}_0^1 yz\,dz)\Big|_{y=0}^{y=x} \;=\; (x+\mathsf{int}_0^1 xz\,dz)-0 \;=\; x+\tfrac{1}{2}xz^2\Big|_{z=0}^{z=1} \;=\; \tfrac{3}{2}x$$

The λ-calculus formalizes this informal notion and provides a solution in the form of safe substitution.

3 Rewrite Rules

3.1 β-reduction

The *β-reduction* rule is the main rule by which evaluation takes place in the λ-calculus:

$$(\lambda x.\,e_1)\,e_2 \;\stackrel{1}{\to}\; e_1\,\{e_2/x\}.$$

An instance of the left-hand side is called a *redex* and the corresponding instance of the right-hand side is called the *contractum*. In the pure λ-calculus, a β-reduction may be performed at any time to any subterm that is a redex of the β-rule. The rule is applied by replacing the redex by its corresponding contractum. For example,

$$\lambda x.\,\underbrace{(\lambda y.\,y)\,x}_{\beta\text{ redex}} \;\stackrel{1}{\to}\; \lambda x.\,x$$

Here the subterm $(\lambda y.\,y)\,x$, which is a redex of the β-rule, is replaced by its contractum x, which is $y\,\{x/y\}$.

3.2 α-conversion

In $\lambda x.\,xz$, the name of the bound variable x does not really matter. This term is semantically the same as $\lambda y.\,yz$. A renaming like this is known as an *α-conversion* or *α-reduction*. In an α-conversion, the new bound variable must be chosen so as to avoid capture. If a term α-reduces to another term, then the two terms are said to be *α-equivalent*. This defines an equivalence relation on the set of terms, denoted $e_1 =_\alpha e_2$.

Recall the definition of free variables $FV(e)$ of a term e. In general we have

$$\lambda x.\,e \;=_\alpha\; \lambda y.\,e\{y/x\} \text{ if } y \notin FV(e).$$

The proviso $y \notin FV(e)$ is to avoid the capture of any free occurrences of y in e as a result of the renaming. The y substituted for x cannot be captured by a binding operator λy already in e because safe substitution $e\{y/x\}$ would not let that happen—it would rename the bound variable accordingly.

3.3 Stoy Diagrams and de Bruijn Indices

There are some other ways to represent bindings without variables. One way is with *Stoy diagrams* (after Joseph E. Stoy). We can create a Stoy diagram for a closed term in the following manner. Instead of writing a term with variable names, we replace each occurrence of a variable with a dot, then connect that dot to the binding operator that binds that variable. For example, $\lambda x.\,(\lambda y.\,(\lambda x.\,xy)\,x)\,x$ becomes the Stoy diagram shown in Fig. 2. Two terms are α-equivalent iff they have the same Stoy diagram, so there is no need for α-conversion.

Figure 2: A Stoy diagram

A related approach is to represent variables using *de Bruijn indices* (after Nicolaas Govert de Bruijn). Here we replace each occurrence of a variable with a natural number indicating the binding operator that binds it. The variable is replaced by the number n if the binding operator that binds it has the n-th smallest scope (counting from 0) among all scopes containing that occurrence of the variable. The example above becomes $\lambda.\,(\lambda.\,(\lambda.\,0\,1)\,1)\,0$. As with Stoy diagrams, two terms are α-equivalent iff their de Bruijn terms are identical.

3.4 η-reduction

Consider the two terms e and $\lambda x.\,ex$, where $x \notin FV(e)$. If these two terms are both applied to an argument e', then they will both reduce to $e\,e'$. Formally,

$$(\lambda x.\,e_1 x)\,e_2 \;\xrightarrow{1}\; e_1\,e_2 \text{ if } x \notin FV(e_1).$$

This says that e and $\lambda x.\,ex$ behave the same way as functions and should be considered equal. Another way of stating this is that e and $\lambda x.\,ex$ behave the same way in all contexts of the form $[\cdot]\,e'$.

This gives rise to a reduction rule called *η-reduction*:

$$\lambda x.\,ex \;\xrightarrow{\eta}\; e \;\text{ provided } x \notin FV(e).$$

The reverse operation, called *η-expansion*, is practical as well.

In practice, η-expansion is used to delay divergence by trapping expressions inside λ-terms. Such terms are sometimes called *thunks*.

3.5 Values and Ω

In the classical λ-calculus, a *value* is just a term containing no β-redexes. Such a term is said to be in *normal form*; no further β-reductions can be applied. Starting from some λ-term, we might perform β-reductions as long as possible, seeking to produce a value. We write $e \Downarrow v$ when a sequence of β-reductions starting with e produces the value v. Does this always happen eventually? Let us define an expression we will call Ω:

$$\Omega \;\triangleq\; (\lambda x.\,xx)\,(\lambda x.\,xx)$$

What happens when we try to evaluate it?

$$\Omega \;=\; (\lambda x.\,xx)\,(\lambda x.\,xx) \;\xrightarrow{1}\; (xx)\{(\lambda x.\,xx)/x\} \;=\; \Omega$$

We have just coded an infinite loop! Thus the term Ω has no value.

4 Confluence

A λ-term in general may have many redexes. A *reduction strategy* is a rule for determining which redex to reduce next. We can think of a reduction strategy as a mechanism for resolving the nondeterminism. In

the classical λ-calculus, no reduction strategy is specified; any redex may be chosen to be reduced next, so the process is nondeterministic. One sequence of reductions may terminate, but another may not. It is even conceivable that different terminating reduction sequences result in different values. Luckily, it turns out that the latter cannot happen.

It turns out that the λ-calculus is *confluent* (also known as the *Church–Rosser* property) under α- and β-reductions. Confluence says that if e reduces by some sequence of reductions to e_1, and if e also reduces by some other sequence of reductions to e_2, then there exists an e_3 such that both e_1 and e_2 reduce to e_3, as illustrated in Fig. 3.

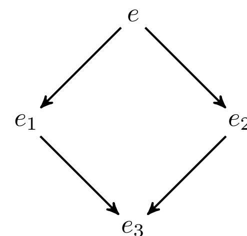

Figure 3: Confluence

It follows that normal forms are unique up to α-equivalence. If $e \Downarrow v_1$ and $e \Downarrow v_2$, where v_1 and v_2 are in normal form, then by confluence they must be α-equivalent. Moreover, regardless of the order of previous reductions, it is always possible to get to the unique normal form if it exists.

However, note that it is still possible for a reduction sequence not to terminate, even if the term has a normal form. For example, $(\lambda xy.\,y)\,\Omega$ has a nonterminating reduction sequence

$$(\lambda xy.\,y)\,\Omega \xrightarrow{1} (\lambda xy.\,y)\,\Omega \xrightarrow{1} \cdots$$

by applying β-reductions to Ω, but also has a terminating reduction sequence, namely

$$(\lambda x.\,\lambda y.\,y)\,\Omega \xrightarrow{1} \lambda y.\,y$$

by applying a β-reduction to the whole term. It may be difficult to determine the most efficient way to expedite termination. But even if we get stuck in a loop, the confluence property guarantees that it is always possible to get unstuck, provided the normal form exists.

References

[1] H. P. Barendregt. *The Lambda Calculus, Its Syntax and Semantics.* North-Holland, 2nd edition, 1984.

[2] Henk P. Barendregt and Jan Willem Klop. Applications of infinitary lambda calculus. *Inf. and Comput.*, 207(5):559–582, 2009.

[3] James Gosling, Bill Joy, Jr. Guy L. Steele, and Gilad Bracha. *The Java Language Specification.* Prentice Hall, 3rd edition, 2005.

[4] John E. Hopcroft and Richard M. Karp. A linear algorithm for testing equivalence of finite automata. Technical Report 71-114, University of California, 1971.

[5] Jean-Baptiste Jeannin. Capsules and closures. *Electron. Notes Theor. Comput. Sci.*, 276:191–213, September 2011.

[6] Jean-Baptiste Jeannin and Dexter Kozen. Capsules and separation. In Nachum Dershowitz, editor, *Proc. 27th ACM/IEEE Symp. Logic in Computer Science (LICS'12)*, pages 425–430, Dubrovnik, Croatia, June 2012. IEEE.

[7] Jean-Baptiste Jeannin and Dexter Kozen. Computing with capsules. In Martin Kutrib, Nelma Moreira, and Rogério Reis, editors, *Proc. Conf. Descriptional Complexity of Formal Systems (DCFS 2012)*, volume 7386 of *Lecture Notes in Computer Science*, pages 1–19, Braga, Portugal, July 2012. Springer.

[8] J. W. Klop and R. C. de Vrijer. Infinitary normalization. In S. Artemov, H. Barringer, A. S. d'Avila Garcez, L. C. Lamb, and J. Woods, editors, *We Will Show Them: Essays in Honour of Dov Gabbay*, volume 2, pages 169–192. College Publications, 2005.

[9] Peter J. Landin. The mechanical evaluation of expressions. *Computer Journal*, 6(4):308–320, 1964.

[10] Saunders MacLane. *Categories for the Working Mathematician.* Springer-Verlag, New York, 1971.

[11] John McCarthy. History of LISP. In Richard L. Wexelblat, editor, *History of programming languages I*, pages 173–185. ACM, 1981.

[12] Robin Milner and Mads Tofte. Co-induction in relational semantics. *Theoretical Computer Science*, 87(1):209–220, 1991.

[13] Eugenio Moggi. Notions of computation and monads. *Information and Computation*, 93(1), 1991.

[14] Benjamin C. Pierce. *Types and Programming Languages.* MIT Press, 2002.

[15] Robert Pollack. Polishing up the Tait–Martin-Löf proof of the Church–Rosser theorem. In *Proc. De Wintermöte '95.* Department of Computing Science, Chalmers University, Göteborg, Sweden, January 1995.

[16] Masako Takahashi. Parallel reductions in λ-calculus (revised version). *Information and Computation*, 118(1):120–127, April 1995.

[17] Robert Endre Tarjan. Efficiency of a good but not linear set union algorithm. *J. ACM*, 22(2):215–225, 1975.

[18] Philip Wadler. Monads for functional programming. In M. Broy, editor, *Marktoberdorf Summer School on Program Design Calculi*, volume 118 of *NATO ASI Series F: Computer and systems sciences.* Springer Verlag, August 1992.

[19] Glynn Winskel. *The Formal Semantics of Programming Languages.* MIT Press, 1993.