1 Reduction Strategies

In general there may be many possible β-reductions that can be performed on a given λ-term. How do we choose which one to perform next? Does it matter?

A specification that tells which of the possible β-reductions to perform next is called a *reduction strategy*. The λ-calculus does not specify a reduction strategy; it is *nondeterministic*. A reduction strategy is needed in real programming languages to resolve the nondeterminism.

Let us define a *value* to be either a primitive value (such as a constant for an integer or Boolean value or a built-in function) or a closed λ-expression for which no β-reductions are possible, given our chosen reduction strategy. For example, $\lambda x.\, x$ would always be a value, whereas $(\lambda x.\, x)\, 1$ would most likely not be.

Most functional programming languages use a reduction strategy known as *call by value* (CBV). (A notable exception is Haskell.) Under CBV, functions may only be called on values; that is, the arguments must be fully evaluated. Thus the β-reduction step $(\lambda x.\, e_1)\, e_2 \xrightarrow{1} e_1\{e_2/x\}$ only applies if e_2 is a value. Here is an example of a CBV evaluation sequence, where 3 and succ (the successor function) are primitive constants.

$$(\lambda x.\, \mathsf{succ}\, x)\, ((\lambda y.\, \mathsf{succ}\, y)\, 3) \quad \xrightarrow{1} \quad (\lambda x.\, \mathsf{succ}\, x)\, (\mathsf{succ}\, 3) \quad \xrightarrow{1} \quad (\lambda x.\, \mathsf{succ}\, x)\, 4 \quad \xrightarrow{1} \quad \mathsf{succ}\, 4 \quad \xrightarrow{1} \quad 5.$$

An alternative strategy is *call by name* (CBN). Under CBN, we defer evaluation of arguments until as late as possible, applying reductions from left to right within the expression. Here is the same term evaluated under CBN.

$$(\lambda x.\, \mathsf{succ}\, x)\, ((\lambda y.\, \mathsf{succ}\, y)\, 3) \quad \xrightarrow{1} \quad \mathsf{succ}\, ((\lambda y.\, \mathsf{succ}\, y)\, 3) \quad \xrightarrow{1} \quad \mathsf{succ}\, (\mathsf{succ}\, 3) \quad \xrightarrow{1} \quad \mathsf{succ}\, 4 \quad \xrightarrow{1} \quad 5.$$

2 Structured Operational Semantics (SOS)

Let's formalize CBV for the pure λ-calculus. First, we need to define the values of the language. These are simply the closed lambda terms:

$$v \quad ::= \quad \lambda x.\, e$$

Next, we can write *inference rules* to specify when reductions are allowed:

$$\frac{}{(\lambda x.\, e)\, v \xrightarrow{1} e\{v/x\}}\ (\beta\text{-reduction}) \qquad \frac{e_1 \xrightarrow{1} e_1'}{e_1\, e_2 \xrightarrow{1} e_1'\, e_2} \qquad \frac{e \xrightarrow{1} e'}{v\, e \xrightarrow{1} v\, e'} \tag{1}$$

This is a simple operational semantics for a programming language based on the λ-calculus. An operational semantics is a language semantics that describes how to run the program. This can be done through informal human-language text, as in the Java Language Specification [1], or through more formal rules.

Rules of the form (1) are known as a Structural Operational Semantics (SOS). They define evaluation as the result of applying the rules to transform the expression. The rules are typically defined in terms of the structure of the expression being evaluated.

As defined above, CBV evaluation is *deterministic*: there is at most one evaluation rule that applies in any situation (we will prove this later).

This kind of operational semantics is known as a *small-step* semantics because it describes only one step at a time. An alternative is a *big-step* (or *large-step*) semantics that describes the entire evaluation of the program to a final value.

We will see other kinds of semantics later in the course, such as *axiomatic semantics*, which describes the behavior of a program in terms of the observable properties of the input and output states, and *denotational semantics*, which translates a program into an underlying mathematical representation.

CBN has slightly simpler rules:

$$\frac{}{(\lambda x.\, e_1)\, e_2 \xrightarrow{1} e_1\, \{e_2/x\}}\ (\beta\text{-reduction}) \qquad \frac{e_0 \xrightarrow{1} e_0'}{e_0\, e_1 \xrightarrow{1} e_0'\, e_1}$$

We don't need the rule for evaluating the right-hand side of an application because β-reductions are performed immediately once the left-hand side is a value.

What happens if we try using Ω as a parameter? It depends on the evaluation strategy. Consider

$$(\lambda x.\, \lambda y.\, y)\, \Omega$$

Using the CBV evaluation strategy, we must first reduce Ω. This puts the evaluator into an infinite loop. On the other hand, CBN reduces the term above to $\lambda y.\, y$. CBN has an important property: CBN will not loop infinitely unless every other semantics would also loop infinitely, yet it agrees with CBV whenever CBV terminates successfully.

2.1 Other Reduction Strategies

In *normal order*, the leftmost redex is always reduced first. This is closely related to CBN evaluation, but also allows reductions in the body of a λ-term. Like CBN, it finds a normal form if one exists, albeit not necessarily in the most efficient way. Call-by-value (CBV) is correspondingly related to *applicative order*, where arguments are reduced first.

In the programming language C, the order of evaluation of arguments is not defined by the language; it is implementation-specific. Because of this and the fact that C has side effects, C is not confluent. For example, the value of the expression $(x = 1) + x$ is 2 if the left operand of + is evaluated first, $x + 1$ if the right operand is evaluated first. This makes writing correct C programs more challenging!

The absence of confluence in concurrent imperative languages is one reason that concurrent programming is difficult. In the λ-calculus, confluence guarantees that reductions can be done in parallel without fear of changing the result.

3 Term Equivalence

When are two terms equal? This question is not as simple as it may seem. The strictest definition of equality is syntactic identity, but this is not very interesting or useful. For example, it seems clear that $\lambda x.\, x$ and $\lambda y.\, y$ should be considered equal, as the parameter name is inconsequential. So we might declare two terms equal if they are syntactically identical modulo α-renaming. This is a reasonable definition if we wish to regard λ-terms as *intensional* objects.

As *extensional* objects, however, it does not go far enough. We would like to consider two terms equal if they represent the same function. The terms $\lambda x.\, x$ and $\lambda y.\, y$ certainly represent the same function (the identity), but there are others; for example, $\lambda x.\, (\lambda y.\, y)\, x$. So terms do not have to be α-equivalent to represent the same function. However, note that $\lambda x.\, (\lambda y.\, y)\, x$ reduces to $\lambda x.\, x$ in one β-reduction step applied inside the body of the outer λ-expression. So we might declare two terms equal if they have a common normal form; that is, if they converge to α-equivalent values when reductions are applied. By confluence, this is an equivalence relation. This *normalization* approach is useful for compiler optimization and for checking type equality in some advanced type systems. Unfortunately, it would not work for reduction strategies like CBN and CBV, which do not allow reductions inside the body of a λ-expression.

It would be nice if we could just say that two terms are equivalent if they give equivalent results on equivalent inputs. Unfortunately, this is a circular statement, so it doesn't define anything! It is not even clear that there is a "right" definition.

Another complication is undecidability. It is likely that any reasonable notion of extensional equivalence will be undecidable due to the relationship between the λ-calculus and Turing machines. If we could test equivalence, then we could test equivalence with Ω, which is tantamount to solving the halting problem.

3.1 Contexts and Observational Equivalence

Another approach to the problem of defining equivalence is to say that two terms are equivalent if they behave indistinguishably in any possible context. But what do we mean by "behave indistinguishably"?

For simplicity, let us assume that we are working with an evaluation strategy such as CBV or CBN that is *deterministic*, which means that there is at most one next β-reduction that can be performed. We say that a term e *terminates* or *converges* if there is a finite sequence of reductions

$$e \;\rightarrow\; e' \;\rightarrow\; e'' \;\rightarrow\; \cdots \;\rightarrow\; v$$

leading to a value v. We write $e \Downarrow v$ when this happens, and we write $e \Downarrow$ when $e \Downarrow v$ for some v. The other possibility is that it keeps on reducing forever without ever arriving at a value. When this happens, we say that e *diverges* and write $e \Uparrow$. Because the computation is deterministic, exactly one of these two cases will occur.

With CBN or CBV, there are infinitely many divergent terms. One example is Ω, which was defined in the last lecture. We might consider all divergent terms equivalent, since none of them produce a value.

While we may not have a precise definition of extensional equivalence yet, we can postulate a desirable property: two equivalent terms, when placed in the same context, should either both diverge or both converge and give indistinguishable values. Here a *context* is any term $C[\cdot]$ with a single occurrence of a distinguished special variable, called the *hole*, and $C[e]$ denotes the context $C[\cdot]$ with the hole replaced by the term e. This notion of equivalence is called *observational equivalence.*

More formally, suppose we already have a notion of equivalence $\equiv$ on values. Then we will say that two terms are *observationally equivalent* (with respect to $\equiv$) and write $e_1 \equiv_{\text{obs}} e_2$ iff

- for all contexts $C[\cdot]$, $C[e_1] \Downarrow$ iff $C[e_2] \Downarrow$; and
- if $C[e_1] \Downarrow v_1$ and $C[e_2] \Downarrow v_2$, then $v_1 \equiv v_2$.

In other words, either both $C[e_1]$ and $C[e_2]$ diverge, or both converge and produce equivalent values.

However, note that some terms are values, and for them, equivalence is not necessarily the same as observational equivalence. We could easily have values that are equivalent in the sense of $\equiv$ but are not observationally equivalent. Is it possible to have $\equiv_{\text{obs}}$ and $\equiv$ coincide on values? In other words, does there exist a *fixpoint* of the transformation $\equiv \mapsto \equiv_{\text{obs}}$? If so, is it unique? Even if not, is there a reasonable choice for the definition of extensional equivalence?

The answers to these questions lie in the following facts, none of which are difficult to prove. We leave them as exercises.

Lemma 1. *Let $\equiv$ be an arbitrary equivalence relation on values.*

(i) *The relation* $\equiv_{\text{obs}}$ *is an equivalence relation on terms.*

(ii) *Restricted to values,* $\equiv_{\text{obs}}$ refines $\equiv$*; that is, viewed as sets of ordered pairs,* $\equiv_{\text{obs}}$ *restricted to values is a subset of* $\equiv$*. Thus for any values* v_1 *and* v_2*, if* $v_1 \equiv_{\text{obs}} v_2$*, then* $v_1 \equiv v_2$*.*

(iii) *If* $e_1 \equiv_{\text{obs}} e_2$*, then for all contexts* $C[\cdot]$*,* $C[e_1]\Downarrow$ *iff* $C[e_2]\Downarrow$*.*

(iv) *The transformation* $\equiv\ \mapsto\ \equiv_{\text{obs}}$ *is* monotone *with respect to the refinement relation. That is, if* $\equiv^1$ *refines* $\equiv^2$*, then* $\equiv^1_{\text{obs}}$ *refines* $\equiv^2_{\text{obs}}$*.*

Now we can see that there are several fixpoints of the transformation $\equiv\ \mapsto\ \equiv_{\text{obs}}$; the identity relation and the relation of α-equivalence, for two. This follows from Lemma 1(i) and (ii). For CBV and CBN, there is also a *coarsest* one that is refined by every other fixpoint: define

$$e_1 \equiv_{\Downarrow} e_2 \quad \overset{\triangle}{\iff} \quad \text{for all contexts } C[\cdot],\ C[e_1]\Downarrow \text{ iff } C[e_2]\Downarrow .$$

Theorem 2. *For CBV and CBN, the relation* $\equiv_{\Downarrow}$ *is a fixpoint of the transformation* $\equiv\ \mapsto\ \equiv_{\text{obs}}$*; that is,* $\equiv_{\Downarrow} = (\equiv_{\Downarrow})_{\text{obs}}$*. Moreover, it is the coarsest such fixpoint.*

The relation $\equiv_{\Downarrow}$ may be a reasonable candidate for extensional equivalence. By definition, to check that e_1 and e_2 are observationally equivalent, it is enough to check that e_1 and e_2 both converge or both diverge in any context; it is unnecessary to compare the resulting values in the case of convergence. This is because if the values are not equivalent, one can devise a context in which one converges and the other diverges.

References

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