Lecture Notes on Program Equivalence

15-312: Foundations of Programming Languages
Frank Pfenning

Lecture 24
November 30, 2004

When are two programs equal? Without much reflection one might say that two programs are equal if they evaluate to the same value, or if both of them run forever. This explicitly ignores the issue of effects, and we will continue to think about a pure language until later in this lecture. So, in a pure language the statement above reduces the equality of programs to the equality of values. But when should two values be equal? For example, how about the following two functions.

$$\begin{array}{lcl} id_1 & = & \lambda x.x \\ id_2 & = & \lambda x.x + 0 \end{array}$$

The first observation is that they are both values, so they definitely will not diverge.

Now, id_1 and id_2 return the same integer when applied to an integer, but id_1 has more types than id_2. We conclude from that the we should compare two values at a type. In general, the judgment has the form

$$v \simeq v' : \tau$$

where we assume that $\cdot \vdash v : \tau$ and $\cdot \vdash v' : \tau$. Here we want to ask if

$$(\lambda x.x) \simeq (\lambda x.x + 0) : \mathsf{int} \to \mathsf{int?}$$

The answer to this question depends on our point of view. If we care about efficiency, for example, they are not equal since the left-hand side always takes one fewer step than the right-hand side. If we care about the syntactic form of the function, they are not equal either. On the other hand,

if we only care about the result of the function when applied to all possible arguments, then the two should be considered equal at the given type, since both of them are (mathematically) the identity function on integers.

In this lecture we are concerned with *observational equivalence* between programs: we consider to programs (and values) equal if whatever we can observe about their behavior is identical. In pure functional languages, the only thing you can observe about a program is the final value it returns. But there are further restrictions. For example, we cannot observe the internal structure of functions. In implementations, they have been compiled to machine code—all we see is a token such as `fn` indicating the given value cannot be printed.

If we cannot observe the structure of a function, what can be observe about a function? We can apply it to arguments and observe its result. But this result may again be a function whose structure we cannot see directly. It appears we are moving in a vicious circle, trying to define observational equivalence of functions in terms of itself.

Fortunately, there is a way out. We once again use *types* in order to create order out of chaos. In our example above, the functions $\lambda x.x$ and $\lambda x.x + 0$ should be equal at type $\mathsf{int} \to \mathsf{int}$ because apply both of them to equal arguments of type int will always yield equal results of type int. And values of type int are directly observable—they form a basic data type of our language.

Using this intuition we can now define two relations of observational equivalence for a pure, call-by-value language by simultaneous induction on the structure of a type of the expressions we are comparing. We write $e \Uparrow$ if the evaluation of e does not terminate. We also use the convention that when we write $e \cong e' : \tau$ that $\cdot \vdash e : \tau$ and $\cdot \vdash e' : \tau$ and similarly for values without restating this every time.

$$
\begin{array}{lll}
e \cong e' : \tau & \text{iff} & \text{either } e \Uparrow \text{ and } e' \Uparrow \\
 & & \text{or } e \mapsto^* v \text{ and } e' \mapsto^* v' \text{ with } v \simeq v' : \tau \\
 & & \\
v \simeq v' : \mathsf{int} & \text{iff} & v = v' = n \text{ for an integer } n. \\
v \simeq v' : \mathsf{bool} & \text{iff} & v = v' = \mathsf{true} \text{ or } v = v' = \mathsf{false} \\
v \simeq v' : \tau_1 \to \tau_2 & \text{iff} & \text{for all } v_1 \simeq v_1' : \tau_1 \text{ we have } v\, v_1 \cong v'\, v_1' : \tau_2
\end{array}
$$

The last clause requires careful analysis. Functions are not observable directly, although we can apply them to arguments to observe their result. The case of values of function type can therefore be summarized as: "*Two functions are equal at type $\tau_1 \to \tau_2$ if they deliver equal results of type τ_2 when applied to equal arguments of type τ_1.*" Note that on the right-hand side the types are smaller than on the left-hand side, so the definition is well-founded. It

is also allowed that neither of the two functions terminates when given equal arguments. This follows from comparing the expressions $v\, v_1$ and $v'\, v_1'$ which have to be evaluated first.

We can use this definition to prove our original assertion that $\lambda x.x \simeq \lambda x.x + 0 : \mathsf{int} \to \mathsf{int}$.

$v_1 \simeq v_1' : \mathsf{int}$	Assumption
$v_1 = v_1' = n$ for some integer n	By definition of $\simeq$
$n \simeq n : \mathsf{int}$	By definition of $\simeq$
$(\lambda x.x)\, n \mapsto^* n$	By definition of $\mapsto$
$(\lambda x.x + 0)\, n \mapsto^* n$	By definition of $\mapsto$
$(\lambda x.x)\, n \cong (\lambda x.x + 0)\, n : \mathsf{int}$	By definition of $\cong$
$(\lambda x.x)\, v_1 \cong (\lambda x.x + 0)\, v_1' : \mathsf{int}$	Since $v_1 = v_1' = n$
$(\lambda x.x) \simeq (\lambda x.x + 0) : \mathsf{int} \to \mathsf{int}$	By definition of $\simeq$

In many cases equivalence proofs are not that straightforward, but require considerable effort. As a slightly more complicated example consider

$$\begin{array}{lcl} id_1 & = & \lambda x.x \\ id_3 & = & \mathsf{rec}\ f.\ \lambda x.\mathsf{if}\ x = 0\ \mathsf{then}\ 0\ \mathsf{else}\ f(x-1) + 1 \end{array}$$

We notice that id_1 and id_3 are in fact *not* equal at type $\mathsf{int} \to \mathsf{int}$ because $id_3\,(-1)$ diverges, while $id_1\,(-1) \mapsto^* -1$. However, when applied to natural numbers, that is, integers greater or equal to 0, then they are observationally equal (both return the argument). In order to capture this we introduce $\mathsf{nat} \leq \mathsf{int}$ under the subset interpretation of subtyping and extend observational equivalence with the clause

$v \simeq v' : \mathsf{nat}$ iff $v = v' = k$ for some $k \geq 0$.

With these definitions we need a lemma, which can be proven by induction on k. For this, we introduce the definition

$$id_3' \quad = \quad \lambda x.\ \mathsf{if}\ x = 0\ \mathsf{then}\ 0\ \mathsf{else}\ id_3\ (x-1) + 1$$

which has the property that $id_3 \mapsto id_3'$ and id_3' is a value. Now we can prove:

For any $k \geq 0$, we have $id_1\, k \cong id_3'\, k : \mathsf{nat}$.

Proof: By induction on k.

Case: $k = 0$. Then $id_1\, 0 \mapsto 0$ and $id_3'\, 0 \mapsto \mathsf{if}\ 0 = 0\ \mathsf{then}\ 0\ \mathsf{else}\ id_3(k-1) + 1 \mapsto^* 0$.

Case: $k = k'+1$. Then $id_1\, k \mapsto k$ and $id_3'\, k \mapsto^* id_3(k-1)+1 \mapsto^* id_3'(k')+1$. By induction hypothesis, $id_3'(k') \cong id_1(k')$ so $id_3'(k') \cong k'$ and $id_3'\, k \mapsto^* k'+1 = k$, which is what we needed to show. ■

From this it follows directly by definition of $\simeq$ that $id_1 = id_3$, since $v_1 \simeq v_1' : \mathsf{nat}$ iff $v_1 = v_1' = k$ for some k and $id_3 \mapsto id_3'$.

Some care must be taken in general to define observational equivalence correctly with respect to what is observable. For example, in a call-by-name language we would have to apply functions to arbitrary expressions, instead of testing them just on values.

It should also be clear that in the presence of effects, be it store effects or control effects, the definition of observational equivalence must be changed substantially to account for the effects.

In the remainder of this lecture we briefly explore the question of equivalence in a setting where we have only effects. In particular, we are no longer interested in termination or the value produced by a computation, but just the externally observable effects it has. This is a fundamental shift in perspective on the notion of computation, but one that is appropriate in the realm of concurrency. For example, we may have server process that never finishes, but forever answers request. It does not return a value (because it never does return), but it interacts with the outside world by receiving requests and sending replies. In this setting, observational equivalence implies that the server answers with equal reply given equal requests. This is a bit imprecise in the setting where we also have non-determinism, that is, a process might evolve in different ways.

For this, we introduce the notion of a *sequential process expression*. Sequential processes can evolve non-deterministically and have externally observable actions, but they do not yet integrate concurrency which is reserved for the next lecture. We start with (observable) actions α which, at present consist either of names a (eventually denoting an input action) and co-names $\bar{a}$ (eventually denoting an output action). A sequential process expression P is defined by the following grammar.

$$P \quad ::= \quad A \mid \alpha_1.P_1 + \cdots + \alpha_n.P_n$$

We write 0 for a sum of zero elements; it corresponds to a process that has terminated (it can take no further actions). Note that "." is not related to variable binding here, it simply separates the prefix α from the process expression P. The *process identifiers* A are defined by, possibly recursive equations

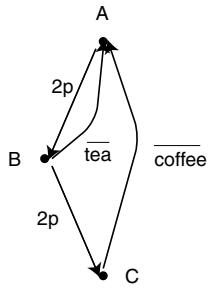
$$A \stackrel{\text{def}}{=} P_A.$$

Sequential process expression evolve in a rather straightforward way. We can unfold a definition of a process identifier, or we can select one non-deterministically from a sum. When such an action is taken, the result is observable. We define a single-step judgment $P \xrightarrow{\alpha} P'$ meaning that P transitions in one step to P' exhibiting action α.

$$\frac{}{M + \alpha.P + N \xrightarrow{\alpha} P}\ \text{Sum} \qquad \frac{(A \stackrel{\text{def}}{=} P_A) \quad P_A \xrightarrow{\alpha} P'}{A \xrightarrow{\alpha} P'}\ \text{Def}$$

The Sum rule non-deterministically selects an element of a sum and exhibits action α. Because of the syntax of the language, we cannot replace a part of the sum. We write M and N for sums.

An examples, consider a tea and coffee vending machine with the following informal behavior: if we put in twopence[1] we can obtain tea by pusing an appropriately labeled button, or we can deposit 2 more pennies and the obtain coffee. This machine can be depicted as



and described as a sequential process as follows:

$$A \stackrel{\text{def}}{=} 2\text{p}.(\overline{\text{tea}}.A + 2\text{p}.\overline{\text{coffee}}.A)$$

The vending machine has three states: an initial state A (in which it only waits for the input of 2p), a state B where we can either get the $\overline{\text{tea}}$, or put in another 2p, and a state C where can only ge the $\overline{\text{coffee}}$. We can make this explicit with this alternative definition

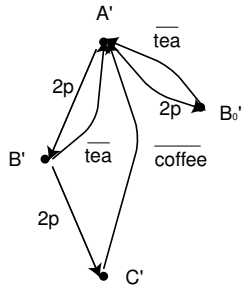
$$\begin{array}{lcl} A & \stackrel{\text{def}}{=} & 2\text{p}.B \\ B & \stackrel{\text{def}}{=} & \overline{\text{tea}}.A + 2\text{p}.C \\ C & \stackrel{\text{def}}{=} & \overline{\text{coffee}}.A \end{array}$$

[1]This example is taken from Robin Milner's book on *Communicating and Mobile Processes: the π-Calculus*, Cambridge University Press, 1999.

Now we return to the question of observational equivalence. If we think just about the actions that the vending machine can exhibit, they can be described by the regular expression:

$$(2\mathrm{p} \cdot (\overline{\mathrm{tea}} + 2\mathrm{p} \cdot \overline{\mathrm{coffee}}))^*.$$

However, this regular expression does not characterize the vending machine as it interacts with its environment. In order to see that, consider the following (broken) vending machine.



$$\begin{array}{lcl} A' & \stackrel{\mathrm{def}}{=} & 2\mathrm{p}.B' + 2\mathrm{p}.B'_0 \\ B' & \stackrel{\mathrm{def}}{=} & \overline{\mathrm{tea}}.A' + 2\mathrm{p}.C' \\ B'_0 & \stackrel{\mathrm{def}}{=} & \overline{\mathrm{tea}}.A' \\ C' & \stackrel{\mathrm{def}}{=} & \overline{\mathrm{coffee}}.A' \end{array}$$

In words, this machine differs from the first one as follows: when we supply it with $2\mathrm{p}$ when in state A', it will non-deterministically go to state B' as before, or go into a new state B'_0 in which we can only obtain $\overline{\mathrm{tea}}$, but not deposit any additional money. Clearly, this machine is broken. However, the sequence of actions it can produce, namely

$$(2\mathrm{p} \cdot (\overline{\mathrm{tea}} + 2\mathrm{p} \cdot \overline{\mathrm{coffee}}) + 2\mathrm{p} \cdot \overline{\mathrm{tea}})^*$$

is exactly the same as for the first machine.

What has gone wrong is the the *reactive* behavior of the system has changed. But this is what we will be interested in when analyzing communicating processes. Here, every input or output will be seen as an interaction with the environment, and then the two vending machines are clearly not equivalent.

In order to capture in what sense they are equivalent we define the notion of *strong simulation*. Let $\mathcal{S}$ be a relation on the states of a process or

between several processes. We say that $\mathcal{S}$ is a *strong simulation* if whenever $P \xrightarrow{\alpha} P'$ and $P \mathcal{S} Q$ then there exists a state Q' such that $Q \xrightarrow{\alpha} Q'$ and $P' \mathcal{S} Q'$. We say that Q strongly simulates P is there exists a strong simulation $\mathcal{S}$ such that $P \mathcal{S} Q$.

For example, the first machine above strongly simulates the second in the sense that there is a strong simulation $\mathcal{S}$ such that $A' \mathcal{S} A$. We write this simulation as $\leq_1$. It is defined by

$$\begin{array}{ll} A' \leq_1 A & \\ B' \leq_1 B & B'_0 \leq_1 B \\ C' \leq_1 C & \end{array}$$

In order to prove that this is a strong simulation we have to verify the conditions in the definition for every transition of the second machine.

Case: $A' \xrightarrow{\text{2p}} B'$ and $A' \leq_1 A$. We have to show there is state Q such that $A \xrightarrow{\text{2p}} Q$ and $B' \leq Q$. $Q = B$ satisfies this condition. We abbreviate this argument in the following case by just showing the relevant transition.

Case: $A' \xrightarrow{\text{2p}} B'_0$ and $A' \leq_1 A$. Then $A \xrightarrow{\text{2p}} B$ and $B'_0 \leq_1 B$.

Case: $B' \xrightarrow{\overline{\text{tea}}} A'$ and $B' \leq_1 B$. Then $B \xrightarrow{\overline{\text{tea}}} A$ and $A' \leq_1 A$.

Case: $B' \xrightarrow{\text{2p}} C'$ and $B' \leq_1 B$. Then $B \xrightarrow{\text{2p}} C$ and $C' \leq_1 C$.

Case: $B'_0 \xrightarrow{\overline{\text{tea}}} A'$ and $B'_0 \leq_1 B$. Then $B \xrightarrow{\overline{\text{tea}}} A$ and $A' \leq_1 A$.

Case: $C' \xrightarrow{\overline{\text{coffee}}} A'$ and $C' \leq_1 C$. Then $C \xrightarrow{\overline{\text{tea}}} A$ and $A' \leq_1 A$.

This covers all cases, so A strongly simulates A'. The perhaps surprising fact is that A' also strongly simulates A, although we need a different relation. We define

$$\begin{array}{l} A \leq_2 A' \\ B \leq_2 B' \\ C \leq_2 C' \end{array}$$

so that B'_0 is not related to any other state. Then $\leq_2$ shows that A' strongly simulates A. Intuitively, this is the case, because the second machine can

simulate every step the first machine can take. It can also exhibit some additional undesired behavior, but this does not matter when we construct a strong simulation.

Now it seems like we have defeated our original purpose, since the two vending machines should not be observationally equivalent, but each one can strongly simulate the other. It turns out that the notion we are interested in is not mutual strong simulation, but *strong bisimulation* which means that there is a *single* relation between the states that acts as a strong simulation in both directions. Under this definition, the two vending machines are not equivalent, because any bi-simulation would have to relate B' and B'_0 to B, but B'_0 could never simulate B because it cannot simulate the transition to C.

In summary, we have isolated the notion of strong bisimulation that we can use to compare the behavior of sequential processes with observable actions and non-deterministic choice. In the next lecture we will make our language of processes richer, allowing for concurrency and interaction.