CS611 Lecture 8 Rule induction, Evaluation contexts, Some concurrency 15 September, 2004

Scribe: Lucja Kot and Michael George Lecturer: Andrew Myers

1 Rule Induction

As a further example of an inductive proof, we give a proof by *rule induction*. Suppose we wish to prove that in lambda calculus with CBV evaluation, the following property holds:

$$e \longrightarrow e' \Rightarrow FV(e) = FV(e') \tag{1}$$

that is, performing a step of the evaluation keeps the set of free variables of a term unchanged. It turns out this statement is not true; we will therefore prove the weaker statement:

$$e \longrightarrow e' \Rightarrow FV(e) \supseteq FV(e') \tag{2}$$

This will allow us to state, for instance, that if we start with a closed term (no free variables), then no free variables are ever added in the evaluation process.

1.1 Proof

We prove the statement by induction, considering each of the possible rules of evaluation that can be applied to e. Recall that the CBV evaluation rules for lambda calculus are:

$$\begin{array}{ccc} \beta\text{-reduction} & \text{(L)} & \text{(R)} \\ \\ \dfrac{}{(\lambda x.\, e_0)v \longrightarrow e_0\{v/x\}} & \dfrac{e_1 \longrightarrow e_1'}{e_1\ e_2 \longrightarrow e_1'\ e_2} & \dfrac{e \longrightarrow e'}{v\ e \longrightarrow v\ e'} \end{array}$$

Define a property P such that:

$$P(e \longrightarrow e') \iff FV(e) \supseteq FV(e')$$

We prove equation 2 by rule induction, showing that whenever P holds on the top line of any evaluation rule, it will hold on the bottom as well.

Case 1 - Rule L

$$\begin{array}{rcl} FV(e_1 e_2) & = & FV(e_1) \cup FV(e_2) \\ & \supseteq & FV(e_1') \cup FV(e_2) \text{ (ind. hypothesis)} \\ & = & FV(e_1' e_2) \end{array}$$

Case 2 - Rule R

$$\begin{array}{rcl} FV(ve) & = & FV(v) \cup FV(e) \\ & \supseteq & FV(v) \cup FV(e') \text{ (ind. hypothesis)} \\ & = & FV(ve') \end{array}$$

Case 3 - β-reduction rule

The proof for this case is a little trickier. We begin as for the other two cases. Let $e = (\lambda x.\, e_0)v$ and $e' = e_0\{v/x\}$. Then:

$$\begin{aligned} FV(e) &= FV(\lambda x.\, e_0) \cup FV(v) \\ &= FV(e_0) \setminus \{x\} \cup FV(v) \end{aligned}$$

But it is not immediately obvious that

$$FV(e_0) \setminus \{x\} \cup FV(v) \supseteq FV(e_0\{v/x\}) \tag{3}$$

which is the result we need to complete this case and the entire proof of equation 2. We therefore need to prove equation 3, which we do by structural induction on e_0.

Case where $e_0 = x$:

$$FV(e_0) \setminus \{x\} \cup FV(v) = FV(v)$$

and

$$FV(e_0\{v/x\}) = FV(v)$$

so equation 3 holds.

Case where $e_0 = y,\ y \neq x$:

$$\begin{aligned} LHS &= FV(y) \setminus \{x\} \cup FV(v) \\ &= FV(v) \cup \{y\} \end{aligned}$$

and

$$\begin{aligned} RHS &= FV(y\{v/x\}) \\ &= \{y\} \end{aligned}$$

So again equation 3 holds. (Note that up to this step, our proof could have worked for statement 1 just as well as for statement 2 above; this step, however, is the one where we can no longer prove strict equality in this manner).

Case where $e_0 = e_1 e_2$:

$$\begin{aligned} LHS &= FV(e_1 e_2) \setminus \{x\} \cup FV(v) \\ &= (FV(e_1) \cup FV(e_2)) \setminus \{x\} \cup FV(v) \\ &= (FV(e_1) \setminus \{x\}) \cup (FV(e_2) \setminus \{x\}) \cup FV(v) \end{aligned}$$

and

$$\begin{aligned} RHS &= FV(e_1\{v/x\}e_2\{v/x\}) \\ &= FV(e_1\{v/x\}) \cup FV(e_2\{v/x\}) \\ &\supseteq (FV(e_1) \setminus \{x\} \cup FV(v)) \cup (FV(e_1) \setminus \{x\} \cup FV(v)) \text{ (inductive hyp. for (3))} \\ &= LHS \end{aligned}$$

and we are done.

Case where $e_0 = \lambda x.\, e_1$:

$$\begin{aligned} LHS &= FV(\lambda x.\, e_1) \setminus \{x\} \cup FV(v) \\ &= FV(e_1) \setminus \{x\} \cup FV(v) \end{aligned}$$

and

$$\begin{aligned} RHS &= FV((\lambda x.\, e_1)\{v/x\}) \\ &= FV((\lambda x.\, e_1) \\ &= FV(e_1) \setminus \{x\} \\ &\subseteq LHS \end{aligned}$$

Case where $e_0 = \lambda y.\, e_1,\ y \neq x$:

$$\begin{aligned} LHS &= FV(\lambda y.\, e_1) \setminus \{x\} \cup FV(v) \\ &= FV(e_1) \setminus \{x, y\} \cup FV(v) \end{aligned}$$

and

$$\begin{aligned} RHS &= FV((\lambda y.\, e_1)\{v/x\}) \\ &= FV(\lambda x.\, e_1\{v/x\}) \\ &= FV(e_1\{v/x\}) \setminus \{y\} \\ &\subseteq (FV(e_1) \setminus \{x\} \cup FV(v)) \setminus \{y\} \text{ (inductive hyp. for (3))} \\ &\subseteq LHS \end{aligned}$$

This completes the proof of equation 3 by structural induction, and consequently also the proof of equation 2 by rule induction.

2 Evaluation Contexts

When we wrote down the small step semantics for the call by value λ-calculus, we really had two classes of rules. There was the β-rule which actually does interesting computation, and there was the left-hand rule and the right-hand rule which only specified the order of computation. It turns out that we often come across this situation when defining languages, so we give these classes names: the "interesting" rules that do computation are called *reductions*, while the other rules are called *evaluation order rules*.

Although the evaluation order rules are important and need to be specified (for example, they distinguish call by name and call by value semantics), in some sense they are all of the same form and are somewhat uninteresting, so we would like a compact representation for them. For this reason we will introduce *evaluation contexts*.

An evaluation context is an expression with a hole, (written $[\bullet]$). It says that you are allowed to reduce any expression appearing in the hole. For example, in call by value,

$$((\lambda x.x)\ [\bullet])((\lambda y.xy)\ (\lambda a.a))$$

is a valid evaluation context, since no matter what we put in the hole, we're allowed to reduce it next. On the other hand,

$$(\lambda x.[\bullet])\ (\lambda y.y)$$

is not a valid call by value evaluation context because $\lambda x.[\bullet]$ is a value and we don't evaluate the innards of a value.

If $E[\bullet]$ is an evaluation context, then we write $E[e]$ for $E[\bullet]$ with e substituted for the hole. This isn't really a lambda abstraction; it operates on the meta-syntactic level. However, this notation allows us to introduce a master evaluation order rule of the form

$$\frac{E[e] \longrightarrow E[e']}{e \longrightarrow e'}$$

and all we have to do to give all of our evaluation order rules is to define $E[\bullet]$. We'll do this with a grammar:

$$E[\bullet] ::= [\bullet] \ \mid\ [\bullet]\ e \ \mid\ v\ [\bullet]$$

With this definition, the master rule gives exactly the left hand rule and the right hand rule for call by value semantics. On the other hand, if we defined

$$E[\bullet] ::= [\bullet] \ \mid\ [\bullet]\ e$$

we would have the evaluation order rules for call by name semantics.

3 Example: Concurrent IMP

Let's say that we wanted to extend our IMP language to express nondeterministic computations. First, we'll augment our syntax to add notation for running two commands in parallel (written $\|$):

$$C ::= (c_1 \parallel c_2) \ \mid\ (c_1; c_2) \ \mid\ \cdots$$

Once this is done, we need to extend the valid evaluation contexts:

$$E[\bullet] ::= ([\bullet]; c) \ \mid\ ([\bullet] \parallel c) \ \mid\ (c \parallel [\bullet])$$

This says that for sequential composition (written ;), we must complete the left hand side before we complete the right hand side, but for parallel composition, we can work on either side independently. Finally, we can add some reduction rules:

$$\overline{\langle \mathbf{skip} \parallel c, \sigma\rangle \longrightarrow \langle c, \sigma\rangle} \qquad \overline{\langle c \parallel \mathbf{skip}, \sigma\rangle \longrightarrow \langle c, \sigma\rangle}$$

Note that it is not straightforward to add concurrency to the big step semantics in this way, because big step semantics discuss entire computations, making it hard to interleave parts of them.