Methods and Formal Models/ Nachum Dershowitz

Lecture 8, 16-5-2000

Notes by: Eden Chlamtac

Recursive Programs

Introduction

In lecture 7, we defined fixpoints, and proved the following result, which is due to Kleene:

Theorem 1 *Every continuous functional B[f] on a complete partial order has a unique least fixpoint*

$$f^{\omega} = lub\{B^i[\Omega]\}$$

We also briefly discussed *computation rules*, which determine the operational semantics of a program by establishing a certain rewriting strategy.

Here we shall discuss the relation between computation rules and least fixpoints. We shall see that any computation rule applied to a *recursive program* (a recursive function definition defined by a functional *B* as in the theorem above) yields a function which is less defined than, or equal to the functional's least fixpoint. We shall also define a class of computation rules which always yield the least fixpoint of a functional.

Recursive Programs and Computation Rules

By a *recursive definition*, or *recursive program*, we mean a program *P* of the form

$$P: \quad f(\overline{x}) \Leftarrow B[f](\overline{x})$$

where B is a continuous functional composed of monotonic base functions and predicates (e.g. *if*, *) and the function variable *f* which are applied to the variables $\overline{x} =< x_1, x_2, ..., x_n >$.

Naturally, we would like a good computation rule to compute a fixpoint of *B[f]* when applied to the computation of *f*. However, we must first examine what we mean by a computation of a recursively defined function.

We shall examine two different types of rewriting rules, *substitution*, and *simplification*. Given some expression α containing occurrences of $f(\overline{x})$, *substitution* replaces some occurrences of $f(\overline{x})$ in α with $B[f](\overline{x})$, whereas *simplification* repeatedly applies the rewriting system *S*, which defines the base functions and predicates, until the expression can no longer be simplified. Assume the rewriting system *S* is such that simplifications can always be made when the value of some sub-expression can be determined (e.g. $0 \cdot x \rightarrow 0$).

Given a program *P* as above over a domain *D*, and an input value $\overline{d} \in (D^+)^n$, we define the *computation of* $f(\overline{d})$ to be a sequence of expressions $\{t_i\}$ derived recursively as follows:

1. $t_0 = f(\overline{d})$
2. For each $i \geq 0$, an intermediate expression t'_i is obtained from t_i by applying substitution according to a computation rule, *C*. Then t_{i+1} is derived from t'_i by simplification.

A *computation rule C* determines which occurences of $f(\overline{x})$ are to be replaced in each substitution step. We take C_P, the function determined by applying a computation rule *C* to the computation of *f*, to be a mapping as follows: Given input $\overline{d} \in (D^+)^n$, if the sequence $\{t_i\}$, defined as above, is finite (i.e. it ends in some $t_k \in D^+$) then we define $C_P = t_k$. Otherwise, $C_P = \omega$.

In lecture 7, we encountered the computation rules *leftmost-innermost ("call-by-value")* and *leftmost-outermost ("call-by-name")*. The functions computed by these rules are denoted f^{LI} (or LI_P) and f^{LO} (or LO_P), respectively. Here are a few examples of other important computation rules:

1. *Parallel-innermost*: Replace all the innermost occurrences of f simultaneosly. The function computed is denoted f^{PI}.
2. *Parallel-outermost*: Replace all the outermost occurrences of f simultaneosly. The function computed is denoted f^{PO}.
3. *Full-substitution*: Replace all occurrences of f simultaneosly. The function computed is denoted f^F.

- Exercise: Consider the following recursive program:

$$P_1 : \; f(x,y) \Leftarrow if\; x = 0\; then\; 2\; else\; f(x-1, f(x+y, y))$$

  What is the least fixpoint of *f*? What are the functions f^{LI} and f^{LO}?

Computation Paths

We will now investigate an alternative method of computation which, as we shall see, is a generalization of the computation sequence method discussed in the previous section.

Given a program *P* defined as before, a *computation path* is a sequence of expressions $\{C_i\}$ where $C_0 = f$, and for each $i \geq 0$, $C_{i+1}[f](\overline{x})$ is derived from $C_i[f](\overline{x})$ by simultaneously replacing certain occurrences of $f(\overline{x})$ with $B[f](\overline{x})$. In contrast to computation sequences, no simplification is performed in computation paths, and neither is there necessarily a fixed computation rule for all substitutions.

- Exercise: Show that $\{C_i[\Omega]\}$ is a chain of functions.

Before examining the relation between computation sequences and computation paths, let us first show the following result, which is an important property of the substitution and simplification rewriting system:

Lemma 1: Let $\alpha[f]$ be any monotonic functional, and *C* be some rewriting rule. Let $\overline{d} \in (D^+)^n$ be an input value for $\alpha[f]$, and let γ be the expression obtained from $\alpha[f](\overline{d})$ by first applying simplification, then substitution according to *C*, and then applying simplification again. Then there exists a rewriting rule *C'* such that applying substitution using *C'*, followed by simplification, yields γ. That is,

$$\begin{tikzcd}
\alpha[f](\overline{d}) \arrow[rr, "S"'] \arrow[d, "C'"'] & & \beta[f](\overline{d}) \arrow[d, "C"] \\
\alpha'[f](\overline{d}) \arrow[dr, "S"'] & & \beta'[f](\overline{d}) \arrow[dl, "S"] \\
& \gamma &
\end{tikzcd}$$

Proof: Let us suppose $\alpha[f]$ contains *m* occurrences of *f*.

Then we may think of α as an m-ary functional, and $\alpha[f^1, ..., f^m]$ as an equivalent expression if we substitute *f* for all f^j. Clearly, after applying simplification, we arrive at some expression $\beta[f^1, ..., f^m]$. We may also order f^j so that applying substitution using computation rule *C* we get $\beta[f^1, ..., f^m](\overline{d}) \xrightarrow[C]{} \beta[B[f^1], .., B[f^i], f^{i+1}, ..., f^m](\overline{d})$. Let *C'* be the computation rule which chooses $f^1, ..., f^i$ when applying substitution to $\alpha[f^1, ..., f^m]$. Let *R* be the rewriting system which, for all $1 \le j \le i$, has the rule $f^j(\overline{x}) \to B[f](\overline{x})$ (indeed, on the right side we now have *f* and not f^j). Recall that *S* is the rewriting system for simplification. Then clearly $R \cup S$ is an orthogonal system, and therefore has the Church-Rosser property. Note that $\alpha'[f^1, ..., f^m](\overline{d}) = \alpha[B[f], ..., B[f], f^{i+1}, ..., f^m](\overline{d})$ and $\beta'[f^1, ..., f^m](\overline{d}) = \beta[B[f], ..., B[f], f^{i+1}, ..., f^m](\overline{d})$ are both obtained from $\alpha[f^1, ..., f^m]$ by applying the rewriting rules in $R \cup S$. Also note that since neither expression contains any f^j, for $1 \le j \le i$, applying the *S* rules (simplification) repeatedly to either expression eventually yields a normal form for $R \cup S$. Since the system is C-R, we have that it is the same normal form for both expressions, namely, γ. To complete the proof it is sufficient to note that substituting f for all $f^1, ..., f^m$, and applying simplification and substitution using *C* and *C'*, as described above, will yield the same results.

Q.E.D.

It is now easy to show a direct correlation between computation sequences and certain computation paths. The following lemma states that for every computation sequence, there is functionally similar computation path. The proof is left as an exercise.

Lemma 2: For any recursive program $P: \quad f(\overline{x}) \Leftarrow B[f](\overline{x})$, element $\overline{d}$ and computation rule *C*, there corresponds a computation path $\{C_i\}$ (of finite or infinite length equal to that of $\{t_i\}$) such that

1. $C_i[f](\overline{d}) \xrightarrow[S]{} t_i$ and
2. $lub\{C_i[\Omega]\}(\overline{d}) \equiv C_P(\overline{d})$

Exercises:

1. Prove Lemma 2. (Hint: for part (1), use induction and Lemma 1)
2. Consider the following program which computes 2^x:

$$P_2 : \quad f(x) \Leftarrow if\ x = 0\ then\ 1\ else\ f(x-1) + f(x-1)$$

What is the computation of $f(2)$ by the leftmost-outermost rule, and what is the corresponding computation path?

Fixpoint Computation Rules

As we mentioned earlier, we are interested in computation rules which always compute a fixpoint of any given recursive definition. We call these *fixpoint computation rules*. We will now show that all computation rules yield functions which are no more defined than the least fixpoint, which from now on will be denoted by f^ω. Later we will define a class of computational rules for which the computed function is always f^ω.

Theorem 2 (Cadiou): For any computational rule *C*, the computed function C_P is less defined than or equal to f^ω; that is, $C_P \sqsubseteq f^\omega$.

Proof: Let $\overline{d} \in (D^+)^n$ be any input for *f*. Let $\{C_i\}$ be a computation path as in Lemma 2. Note that for each *i*, $B^i[\Omega]$ can be obtained from $C_i[\Omega]$ by replacing certain occurences of Ω. Since *B* is monotonic, and from theorem 1, we have

$$\forall i \geq 0.\ C_i[\Omega] \sqsubseteq B^i[\Omega] \sqsubseteq lub\{B^i[\Omega]\} = f^\omega$$

Thus f^ω is an upper bound of$\{C_i[\Omega]\}$. Therefore, combining this result with Lemma 2, we get

$$C_P(\overline{d}) = lub\{Ci[\Omega]\} \sqsubseteq f^\omega(\overline{d})$$

Q.E.D.

The above theorem implies that if a computation rule always yields a fixpoint of a certain recursive definition, then it necessarily yields *the least fixpoint* of that definition. Thus, we could have equivalently defined a fixpoint computation rule to be a computation rule which for any recursive definition yields the least fixpoint. We now define a general class of computation rules which are all fixpoint computation rules.

Definition: Let $\alpha[f^1, ..., f^m]$ be any monotonic functional. Let *C* be a computational rule which substitutes for $f^1, ..., f^i$ when we take $f^j = f$ for all $1 \leq j \leq m$. This application of *C* is called a *safe substitution* if when substituting $f^j = \Omega$ for $1 \leq j \leq i$ and $f^j = f^\omega$ for $i+1 \leq j \leq m$, we get $\alpha[\Omega, ..., \Omega, f^\omega, ..., f^\omega] \equiv \Omega$. A computation rule is said to be *safe* if it uses only safe substitutions.

Theorem 3 (Vuillemin): Any safe computation rule is a fixpoint computation rule.

Proof: By contradiction.

Let us assume, by way of contradiction, that there exists *C*, a computation rule which is safe but not a fixpoint computation rule. Let *P* be a recursive program, as before, such that $C_P \neq f^\omega$. From theorem 2 we have $C_P \sqsubseteq f^\omega$, therefore, there must be some $\overline{d} \in (D^+)^n$ such that $C_P(\overline{d}) = \omega$ and $(lub\{B^i[\Omega]\}(\overline{d}) =) f^\omega(\overline{d}) \neq \omega$. Thus we have $B^n[\Omega](\overline{d}) \neq \omega$ for some $n \geq 0$.

Let $\{C_i\}$ be the computation path for $f(\overline{d})$ as in lemma 2. We have $C_P(\overline{d}) = \omega$, so clearly $\{C_i\}$ cannot be finite. Otherwise, substituting f^ω for every occurence of *f* in $C_i[f]$, we get (since f^ω is a fixpoint) $\omega \neq f^\omega(\overline{d}) = C_0[f^\omega](\overline{d}) = ... = C_k[f^\omega](\overline{d}) = lub\{Ci[f^\omega]\}(\overline{d}) = Cp(\overline{d}) = \omega$ where *k* is the length of the path.

Therefore the path $\{C_i\}$ must be infinite. Let us briefly introduce the notion of depth of occurrences of *f* in elements of a computation path. For example, if $B[f]$ contains two occurrences of *f*, then in the expression $B[f_1, B[B[f_2, f_3], f_4]]$ we say that f_1 has depth 1, f_2 has depth 3, f_4 has depth 2, etc. Recall that $B^n[\Omega](\overline{d}) \neq \omega$ for some *n*. Since there can be only finitely many substitutions at depth $\leq n$, there must exist some $N \geq 0$ such that $C_{N+1}[f]$ is derived from $C_N[f]$ by substitutions of depth >*n*.

Denote the occurrences of *f* in $C_N[f]$ by $f^1, ..., f^m$, such that $C_N[f] = C_N[f^1, ..., f^m]$ and $C_{N+1}[f] = C_N[B[f^1], ..., B[f^i], f^{i+1}, ..., f^m]$. Note that the depths of $f^1, ..., f^i$ in $C_N[f]$ are all >*n*. Consider the term $C_N[\Omega, ..., \Omega, B^n[\Omega], ..., B^n[\Omega]]$ obtained by substituting Ω for all $f^1, ..., f^i$ and $B^n[\Omega]$ for all $f^{i+1}, ..., f^m$. All occurrences of Ω here are of depth >*n*. Therefore, this expression can be derived from $B^n[\Omega]$ by substituting for certain Ω's. *B[f]* is monotonic, therefore

$$B^n[\Omega] \sqsubseteq C_N[\Omega, ..., \Omega, B^n[\Omega], ..., B^n[\Omega]]$$

$C_N[f]$ is also monotonic, and so

$$C_N[\Omega, ..., \Omega, B^n[\Omega], ..., B^n[\Omega]] \sqsubseteq C_N[\Omega, ..., \Omega, f^\omega, ..., f^\omega]$$

From these two inequations, and since *C* is safe, we get

$$B^n[\Omega] \sqsubseteq C_N[\Omega, ..., \Omega, f^\omega, ..., f^\omega] = \omega$$

which immediately implies $B^n[\Omega] = \omega$, contradicting our assumption.

Q.E.D.

Corollary: The parallel-outermost, full-substitution, and leftmost-outermost[1] rules are safe rules and therefore are fixpoint computation rules.

[1]When the program consists only of strict functions, and if expressions.

Exercises:

1. Why is the leftmost-innermost rule not safe? Illustrate this fact by considering the program P_1 in the first exercise.

$$P_1 : \ f(x,y) \Leftarrow if\ x = 0\ then\ 2\ else\ f(x-1, f(x+y,y))$$

2. The proof of theorem 3 is not very straightforward. Illustrate the direct nature of the relation between safety and fixpoint computation rules by transforming the proof by contradiction to a direct proof.