9 Recursion equations

This chapter explores a simple language **REC** which supports the recursive definition of functions on the integers. The language is applicative in contrast to the imperative language of **IMP**. It can be evaluated in a call-by-value or call-by-name manner. For each mode of evaluation operational and denotational semantics are provided and proved equivalent.

9.1 The language REC

REC is a simple programming language designed to support the recursive definition of functions. It has these syntactic sets:

- numbers $n \in \mathbf{N}$, positive and negative integers,
- variables over numbers $x \in \mathbf{Var}$, and
- function variables $f_1, \ldots, f_k \in \mathbf{Fvar}$.

It is assumed that each function variable $f_i \in \mathbf{Fvar}$ possesses an *arity* $a_i \in \omega$ which is the number of arguments it takes—it is allowed for a_i to be 0 when $f_i()$, consisting of the function f_i of arity 0 applied to the empty tuple, is generally written as just f_i. Terms $t, t_0, t_1, \ldots$ of **REC** have the following syntax:

$$t ::= n \mid x \mid t_1 + t_2 \mid t_1 - t_2 \mid t_1 \times t_2 \mid \textbf{if } t_0 \textbf{ then } t_1 \textbf{ else } t_2 \mid f_i(t_1, \ldots, t_{a_i})$$

For simplicity we shall take boolean expressions to be terms themselves with 0 understood as true and all nonzero numbers as false. (It is then possible to code disjunction as $\times$, negation $\neg b$ as a conditional $\textbf{if } b \textbf{ then } 1 \textbf{ else } 0$ and a basic boolean like the equality test $(t_0 = t_1)$ between terms as $(t_0 - t_1)$—see also Exercise 9.1 below.) We say a term is *closed* when it contains no variables from **Var**.

The functions variables f are given meaning by a *declaration*, which consists of equations typically of the form

$$\begin{aligned} f_1(x_1, \ldots, x_{a_1}) &= t_1 \\ &\vdots \\ f_k(x_1, \ldots, x_{a_k}) &= t_k \end{aligned}$$

where the variables of t_i are included in $x_1, \ldots, x_{a_i}$, for $i = 1, \cdots, k$. The equations can be recursive in that the terms t_i may well contain the function variable f_i and indeed other function variables of $f_1, \ldots, f_k$. Reasonably enough, we shall not allow two defining equations for the same function variable.

In a defining equation

$$f_i(x_1, \ldots, x_{a_i}) = t_i$$

we call the term t_i the *definition* of f_i.

What to take as the operational semantics of **REC** is not so clear-cut. Consider a defining equation

$$f_1(x) = f_1(x) + 1.$$

Computational intuition suggests that $f_1(3)$, say, should evaluate to the same value as $f_1(3) + 1$ which should, in turn, evaluate to the same value as $(f_1(3) + 1) + 1$, and so on. The evaluation of $f_1(3)$ should never terminate. Indeed if the evaluation of $f_1(3)$ were to terminate with an integer value n then this would satisfy the contradictory equation $n = n + 1$. Now suppose, in addition, we have the defining equation

$$f_2(x) = 1.$$

In evaluating $f_2(t)$, for a term t, we have two choices: one is to evaluate the argument t first and once an integer value n is obtained to then proceed with the evaluation of $f_2(n)$; another is to pass directly to the definition of f_2, replacing all occurrences of the variable x by the argument t. The two choices have vastly different effects when taking the argument t to be $f_1(3)$; the former diverges while the latter terminates with result 1. The former method of evaluation, which requires that we first obtain values for the arguments before passing them to the definition is called *call-by-value*. The latter method, where the unevaluated terms are passed directly to the definition, is called *call-by-name*. It is clear that if an argument is needed then it is efficient to evaluate it once and for all; otherwise the same term may have to be evaluated several times in the definition. On the other hand, as in the example of $f_2(f_1(3))$, if the argument is never used its divergence can needlessly cause the divergence of the enclosing term.

Exercise 9.1 Based on your informal understanding of how to evaluate terms in **REC** what do you expect the function s in the following declaration to compute?

$$s(x) = \textbf{if } x \textbf{ then } 0 \textbf{ else } f(x, 0 - x)$$
$$f(x, y) = \textbf{if } x \textbf{ then } 1 \textbf{ else } (\textbf{if } y \textbf{ then } -1 \textbf{ else } f(x - 1, y - 1))$$

Define a function $lt(x, y)$ in **REC** which returns 0 if $x < y$, and a nonzero number otherwise. ❑

for all $n, n_1, \cdots, n_{a_i} \in \mathbf{N}$, and thus by the definition of φ that

$$\begin{array}{rcl} \lambda n_1, \ldots, n_{a_1} \in \mathbf{N}.\ [\![d_1]\!]_{va}\varphi\rho[n_1/x_1, \ldots, n_{a_1}/x_{a_1}] & \sqsubseteq & \varphi_1 \\ & \vdots & \\ \lambda n_1, \ldots, n_{a_k} \in \mathbf{N}.\ [\![d_k]\!]_{va}\varphi\rho[n_1/x_1, \ldots, n_{a_k}/x_{a_k}] & \sqsubseteq & \varphi_k. \end{array}$$

But this makes φ a prefixed point of F as claimed, thus ensuring $\delta \sqsubseteq \varphi$.

Finally, letting t be a closed term, we obtain

$$\begin{array}{rcl} [\![t]\!]_{va}\delta\rho = \lfloor n \rfloor & \Rightarrow & [\![t]\!]_{va}\varphi\rho = \lfloor n \rfloor \\ & & \text{by monotonicity of } [\![t]\!]_{va} \text{ given by Lemma 9.3} \\ & \Rightarrow & t \rightarrow^{d}_{va} n \\ & & \text{from (1) in the special case of an empty list of variables.} \square \end{array}$$

Theorem 9.8 *For t a closed term, n a number, and ρ an arbitrary environment*

$$[\![t]\!]_{va}\delta\rho = \lfloor n \rfloor \quad \textit{iff} \quad t \rightarrow^{d}_{va} n.$$

Proof: Combine the two previous lemmas. $\square$

9.5 Operational semantics of call-by-name

We give rules to specify the evaluation of closed terms in **REC** under call-by-name. Assume a declaration d consisting of defining equations

$$\begin{array}{rcl} f_1(x_1, \ldots, x_{a_1}) & = & d_1 \\ & \vdots & \\ f_k(x_1, \ldots, x_{a_k}) & = & d_k \end{array}$$

The evaluation with call-by-name is formalised by a relation $t \rightarrow^{d}_{na} n$ meaning that the closed term t evaluates under call-by-name to the integer value n. The rules giving this

evaluation relation are as follows:

$$n \to^d_{na} n$$

$$\frac{t_1 \to^d_{na} n_1 \quad t_2 \to^d_{na} n_2}{t_1 \ \mathbf{op}\ t_2 \to^d_{na} n_1\ op\ n_2}$$

$$\frac{t_0 \to^d_{na} 0 \quad t_1 \to^d_{na} n_1}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \to^d_{na} n_1}$$

$$\frac{t_0 \to^d_{na} n_0 \quad t_2 \to^d_{na} n_2 \quad n_0 \not\equiv 0}{\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2 \to^d_{na} n_2}$$

$$\frac{d_i[t_1/x_1, \ldots, t_{a_i}/x_{a_i}] \to^d_{na} n}{f_i(t_1, \ldots, t_{a_i}) \to^d_{na} n}$$

The only difference with the rules for call-by-value is the last, where it is not necessary to evaluate arguments of a function before applying it. Again, the evaluation relation is deterministic:

Proposition 9.9 *If* $t \to^d_{na} n_1$ *and* $t \to^d_{na} n_2$ *then* $n_1 \equiv n_2$.

Proof: By a routine application of rule induction. □

9.6 Denotational semantics of call-by-name

As for call-by-value, a term will be assigned a meaning as a value in $\mathbf{N}_\perp$ with respect to environments for variables and function variables, though the environments take a slightly different form. This stems from the fact that in call-by-name functions do not necessarily need the prior evaluation of their arguments. An environment for variables is now a function

$$\rho : \mathbf{Var} \to \mathbf{N}_\perp$$

and we will write $\mathbf{Env}_{na}$ for the cpo

$$[\mathbf{Var} \to \mathbf{N}_\perp]$$

of such environments. On the other hand, an environment for function variables $f_1, \ldots, f_k$ consists of $\varphi = (\varphi_1, \ldots, \varphi_k)$ where each

$$\varphi_i : \mathbf{N}_\perp^{a_i} \to \mathbf{N}_\perp$$

is a continuous function for $i = 1, \ldots, k$; we write $\mathbf{Fenv}_{na}$ for

$$[\mathbf{N}_\perp^{a_1} \to \mathbf{N}_\perp] \times \cdots \times [\mathbf{N}_\perp^{a_k} \to \mathbf{N}_\perp]$$

the cpo of environments for function variables.

Now, we can go ahead and define $[\![t]\!]_{na} : \mathbf{Fenv}_{na} \to [\mathbf{Env}_{na} \to \mathbf{N}_\perp]$, the denotation of a term t by structural induction:

$$\begin{aligned}
[\![n]\!]_{na} &= \lambda\varphi\lambda\rho.\ \lfloor n \rfloor \\
[\![x]\!]_{na} &= \lambda\varphi\lambda\rho.\ \rho(x) \\
[\![t_1 \ \mathbf{op}\ t_2]\!]_{na} &= \lambda\varphi\lambda\rho.\ [\![t_1]\!]_{na}\varphi\rho\ op_\perp\ [\![t_2]\!]_{na}\varphi\rho \\
&\quad \text{where } \mathbf{op} \text{ is } +, -, \text{ or } \times \\
[\![\mathbf{if}\ t_0\ \mathbf{then}\ t_1\ \mathbf{else}\ t_2]\!]_{na} &= \lambda\varphi\lambda\rho.\ \mathit{Cond}([\![t_0]\!]_{na}\varphi\rho, [\![t_1]\!]_{na}\varphi\rho, [\![t_2]\!]_{na}\varphi\rho) \\
[\![f_i(t_1, \ldots, t_{a_i})]\!]_{na} &= \lambda\varphi\lambda\rho.\ \varphi_i([\![t_1]\!]_{na}\varphi\rho, \ldots, [\![t_{a_i}]\!]_{na}\varphi\rho)
\end{aligned}$$

Again, the semantic function is continuous, and its result in an environment is independent of assignments to variables not in the term:

Lemma 9.10 *Let t be a term of* **REC**. *The denotation $[\![t]\!]_{na}$ is a continuous function* $\mathbf{Fenv}_{na} \to [\mathbf{Env}_{na} \to \mathbf{N}_\perp]$.

Proof: By structural induction using the results of Section 8.4. □

Lemma 9.11 *For all terms t of* **REC**, *if environments $\rho, \rho' \in \mathbf{Env}_{na}$ yield the same result on all variables which appear in t then, for any $\varphi \in \mathbf{Fenv}_{na}$,*

$$[\![t]\!]_{na}\varphi\rho = [\![t]\!]_{na}\varphi\rho'.$$

In particular, the denotation $[\![t]\!]_{na}\varphi\rho$ of a closed term t is independent of the environment ρ.

Proof: A straightforward structural induction on terms t. □

$$\mathit{Cond}(z_0, z_1, z_2) = \begin{cases} z_1 & \text{if} \quad z_0 = \lfloor 0 \rfloor, \\ z_2 & \text{if} \quad z_0 = \lfloor n \rfloor \text{ for some } n \in \mathbf{N} \text{ with } n \neq 0, \\ \perp & \text{otherwise} \end{cases}$$

A declaration d determines a particular function environment. Let d consist of the defining equations

$$\begin{aligned} f_1(x_1,\ldots,x_{a_1}) &= d_1 \\ &\vdots \\ f_k(x_1,\ldots,x_{a_k}) &= d_k. \end{aligned}$$

Define $F : \mathbf{Fenv}_{na} \to \mathbf{Fenv}_{na}$ by taking

$$\begin{aligned} F(\varphi) = (&\lambda z_1,\ldots,z_{a_1} \in \mathbf{N}_\perp.\ [\![d_1]\!]_{na}\varphi\rho[z_1/x_1,\ldots,z_{a_1}/x_{a_1}], \\ &\vdots \\ &\lambda z_1,\ldots,z_{a_k} \in \mathbf{N}_\perp.\ [\![d_k]\!]_{na}\varphi\rho[z_1/x_1,\ldots,z_{a_k}/x_{a_k}]). \end{aligned}$$

As in the call-by-value case (see Section 9.4), the operation of updating environments is definable in the metalanguage of Section 8.4. By the general arguments of Section 8.4, F is continuous, and so has a least fixed point $\delta = fix(F)$.

Example: To see how the denotational semantics captures the call-by-name style of evaluation, consider the declaration:

$$\begin{aligned} f_1 &= f_1 + 1 \\ f_2(x) &= 1 \end{aligned}$$

According to the denotational semantics for call-by-name, the effect of this declaration is that f_1, f_2 are denoted by $\delta = (\delta_1, \delta_2) \in \mathbf{N}_\perp \times [\mathbf{N}_\perp \to \mathbf{N}_\perp]$ where

$$\begin{aligned} (\delta_1,\delta_2) &= \mu\varphi.\ ([\![f_1+1]\!]_{na}\varphi\rho,\ \lambda z \in \mathbf{N}_\perp.\ [\![1]\!]_{na}\varphi\rho[z/x]) \\ &= \mu\varphi.\ (\varphi_1 +_\perp \lfloor 1\rfloor,\ \lambda z \in \mathbf{N}_\perp.\ \lfloor 1\rfloor) \\ &= (\perp, \lambda z \in \mathbf{N}_\perp.\ \lfloor 1\rfloor) \end{aligned}$$

It is simple to verify that the latter is the required least fixed point. Thus

$$[\![f_2(f_1)]\!]_{na}\delta\rho = \delta_2(\delta_1) = \lfloor 1\rfloor.$$

□

We can expect that

$$[\![t]\!]_{na}\delta\rho = \lfloor n\rfloor \quad \text{iff} \quad t \to^d_{na} n$$

whenever t is a closed term. Indeed we do have this equivalence between the denotational and operational semantics.

9.8 Local declarations

From the point of view of a programming language **REC** is rather restrictive. In particular a program of **REC** is essentially a pair consisting of a term to be evaluated together with a declaration to determine the meaning of its function variables. Most functional programming languages would instead allow programs in which function variables are defined as they are needed, in other words they would allow local declarations of the form:

$$\textbf{let rec } f(x_1, \cdots, x_{a_1}) = d \textbf{ in } t.$$

This provides a recursive definition of f with respect to which the term t is evaluated. The languages generally support simultaneous recursion of the kind we have seen in declarations and allow more general declarations as in

$$\begin{array}{llll} \textbf{let} & \textbf{rec} & f_1(x_1, \cdots, x_{a_1}) & = & d_1 \textbf{ and} \\ & & \vdots & & \\ & & f_k(x_1, \cdots, x_{a_k}) & = & d_k \\ \textbf{in} & t & & & \end{array}$$

This simultaneously defines a tuple of functions $f_1, \cdots, f_k$ recursively.

To understand how one gives a denotational semantics to such a language, consider the denotation of

$$S \equiv \textbf{let rec } A \Leftarrow t \textbf{ and } B \Leftarrow u \textbf{ in } v$$

where A and B are assumed to be distinct function variables of arity 0. For definiteness assume evaluation is call-by-name. The denotation of S in a function environment $\varphi \in \mathbf{Fenv}_{na}$ and environment for variables $\rho \in \mathbf{Env}_{na}$ can be taken to be

$$[\![S]\!]\varphi\rho = [\![v]\!]\varphi[\alpha_0/A, \beta_0/B]\rho$$

where (α_0, β_0) is the least fixed point of the continuous function

$$(\alpha, \beta) \mapsto ([\![t]\!]\varphi[\alpha/A, \beta/B]\rho, [\![u]\!]\varphi[\alpha/A, \beta/B]\rho).$$

Exercise 9.17 Write down a syntax extending **REC** which supports local declarations. Try to provide a denotational semantics for the extended language under call-by-name. How would you modify your semantics to get a semantics in the call-by-value case? □

In fact, perhaps surprisingly, the facility of simultaneous recursion does not add any expressive power to a language which supports local declarations of single functions,

though it can increase efficiency. For example, the program S above can be replaced by

$$T \equiv \begin{array}{l} \textbf{let rec } B \Leftarrow (\textbf{let rec } A \Leftarrow t \textbf{ in } u) \\ \textbf{in}(\textbf{let rec } A \Leftarrow t \textbf{ in } v). \end{array}$$

where A and B are assumed to be distinct function variables of arity 0. The proof that this is legitimate is the essential content of Bekić's Theorem, which is treated in the next chapter.

9.9 Further reading

Alternative presentations of the language and semantics of recursion equations can be found in [59], [21], [13] and [58](the latter is based on [13]) though these concentrate mainly on the call-by-name case. Zohar Manna's book [59] incorporates some of the thesis work of Jean Vuillemin on recursion equations [99]. This chapter has been influenced by some old lecture notes of Robin Milner, based on earlier notes of Gordon Plotkin, (though the proofs here are different). The proof in the call-by-value case is like that in Andrew Pitts' Cambridge lecture notes [75]. The operational semantics for the language extended by local declarations can become a bit complicated, as, at least for static binding, it is necessary to carry information about the environment at the time of declaration—see [101] for an elementary account.