Chapter 5: Recursive and Recursively Enumerable Languages

In this chapter, we will study a universal programming language, which we will use to define the recursive and recursively enumerable languages.

We will see that:

- the context-free languages are a proper subset of the recursive languages,
- the recursive languages are a proper subset of the recursively enumerable languages, and
- there are languages that are not recursively enumerable.

Furthermore, we will learn that there are problems, like the halting problem (the problem of determining whether a program halts when run on a given input), that can't be solved by programs.

Introduction

Traditionally, one uses Turing machines for the universal programming language. Turing machines are finite automata that manipulate infinite tapes.

Turing machines are rather far-removed from conventional programming languages, and are hard to build and reason about.

Instead, we will work with a simple functional programming language that has explicit support for formal language symbols and strings.

This language will have the same power as Turing machines, but will be much easier to program in and reason about than Turing machines.

5.1: Programs and Recursive and Recursively Enumerable Languages

In this section, we introduce our functional programming language, and then use it to define the recursive and recursively enumerable languages.

In contrast to Standard ML, our programming language is *dynamically* typed. I.e., all type errors happen at runtime; there is no typechecking phase.

Programs are certain trees (see Section 1.3). The set of all programs is defined via an inductive definition.

The details are in the book. In the slides, we're going to simply work with examples.

Example Program

For example, consider the program:

```
letSimp(equal,
        lam(p,
            calc(isZero,
                 calc(compare, var(p)))),
        app(var(equal),
            pair(str(0110), str(0110))))
```

The let expression declares the variable equal to be equal to the anonymous function that takes in an argument p (which will have to consist of a pair), and compares the components of p for equality. The body of the let expression then applies equal to the pair both of whose elements are the string 0110.

When executed, the program will evaluate to the boolean constant **const**(true).

Program Syntax

Lists and trees can be represented using pairs. E.g., $\mathbf{pair}(pr_1, \mathbf{pair}(pr_2, \mathbf{const}(\mathsf{nil})))$ represents the list with first element pr_1 and second element pr_2.

A program is *closed* when it has no free variables, i.e., all of its variables are declared. We write **CP** for the set of all closed programs.

Program Syntax

Programs can also be described as strings over the alphabet consisting of the letters and digits, plus the elements of $\{\langle\text{comma}\rangle, \langle\text{perc}\rangle, \langle\text{tilde}\rangle, \langle\text{openPar}\rangle, \langle\text{closPar}\rangle, \langle\text{less}\rangle, \langle\text{great}\rangle\}$.

For example, the program

$$\textbf{calc}(\text{plus}, \textbf{pair}(\textbf{int}(4), \textbf{int}(-5)))$$

which adds 4 and -5 together, is described by the string

$$\text{calc}\langle\text{openPar}\rangle\text{plus}\langle\text{comma}\rangle\text{pair}\langle\text{openPar}\rangle\text{int}\langle\text{openPar}\rangle 4\langle\text{closPar}\rangle \langle\text{comma}\rangle\text{int}\langle\text{openPar}\rangle\langle\text{tilde}\rangle 5\langle\text{closPar}\rangle\langle\text{closPar}\rangle\langle\text{closPar}\rangle.$$

Every program is described by a unique string, and every string describes at most one program. (E.g., the string $\langle\text{comma}\rangle\langle\text{closPar}\rangle$ doesn't describe a program.)

Programming Language Semantics

We write **Val** for the set of all elements of **CP** that are values, i.e., are completely evaluated: **const**(true), **const**(false), **const**(nil), integers, symbols, strings, pairs both of whose sides are values, and anonymous functions.

Let $\mathbf{Eval} = \{\mathbf{nonterm}, \mathbf{error}\} \cup \{\, \mathbf{norm}\ pr \mid pr \in \mathbf{Val} \,\}$.

The book defines a mathematical function (*not* an algorithm) $\mathbf{eval} \in \mathbf{CP} \to \mathbf{Eval}$ that tries to evaluate a closed program pr, returning:

- **nonterm**, if that evaluation never terminates,
- **error**, if that evaluation results in an error, and
- **norm** pr', if that evaluation results in the value pr'.

Programs in Forlan

The `Prog` module defines the abstract type (in the top-level environment) `prog` of programs, along with a number of functions, including:

```
val input    : string -> prog
val output   : string * prog -> unit
val height   : prog -> int
val size     : prog -> int
val equal    : prog * prog -> bool
val evaluate : prog * int -> unit
val fromStr  : str -> prog
val toStr    : prog -> str
```

Programs in Forlan

Suppose that we put the text

```
letSimp(equal,
        lam(p,
            calc(isZero,
                 calc(compare, var(p)))),
        app(var(equal),
            pair(str(0110), str(0110))))
```

in the file `5.1-equal-prog`. Then we can proceed as follows.

```
- val pr = Prog.input "5.1-equal-prog";
val pr = - : prog
- Prog.height pr;
val it = 4 : int
- Prog.size pr;
val it = 10 : int
```

Programs in Forlan

```
- Prog.evaluate(pr, 2);
intermediate result
"calc(isZero,
      calc(compare, pair(str(0110), str(0110))))"
val it = () : unit
- Prog.evaluate(pr, 3);
intermediate result "calc(isZero, int(0))"
val it = () : unit
- Prog.evaluate(pr, 6);
terminated with value "const(true)"
val it = () : unit
```

Programs in Forlan

```
- val x = Prog.toStr pr;
val x =
  [-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,
   -,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,
   -,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,
   -,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-,-] : str
- Str.output("", x);
letSimp<openPar>equal<comma>lam<openPar>p<comma>calc
<openPar>isZero<comma>calc<openPar>compare<comma>var
<openPar>p<closPar><closPar><closPar><closPar><comma>a
pp<openPar>var<openPar>equal<closPar><comma>pair
<openPar>str<openPar>0110<closPar><comma>str<openPar>0
110<closPar><closPar><closPar><closPar>
val it = () : unit
```

Programs in Forlan

```
- val pr' = Prog.fromStr x;
val pr' = - : prog
- Prog.equal(pr', pr);
val it = true : bool
- Prog.evaluate(Prog.fromString "lam(y, var(x))", 10);
program has free variables: "x"

uncaught exception Error
- val pr'' = Prog.input "";
@ calc(plus, calc(compare, pair(int(3), int(4))))
@ .
val pr'' = - : prog
- Prog.evaluate(pr'', 10);
terminated with error "calc(plus, int(~1))"
val it = () : unit
```

Graphical Editor for Program Trees

The Java program JForlan, can be used to view and edit program trees. It can be invoked directly, or run via Forlan. See the Forlan website for more information.

Parsing in Our Language

We can write a function for parsing a program from a string, where a program pr will be represented as a tree (built using pairs) value $\overline{pr}$. The parser returns $\mathbf{const}(\mathsf{nil})$ to indicate failure.

For example, we can have:

$$\begin{aligned}
\overline{\mathbf{var}(\mathsf{foo})} &= \mathbf{pair}(\mathbf{str}(\mathsf{var}), \mathbf{str}(\mathsf{foo})),\\
\overline{\mathbf{const}(\mathsf{true})} &= \mathbf{pair}(\mathbf{str}(\mathsf{const}), \mathbf{const}(\mathsf{true})),\\
\overline{\mathbf{const}(\mathsf{false})} &= \mathbf{pair}(\mathbf{str}(\mathsf{const}), \mathbf{const}(\mathsf{false})),\\
\overline{\mathbf{const}(\mathsf{nil})} &= \mathbf{pair}(\mathbf{str}(\mathsf{const}), \mathbf{const}(\mathsf{nil})),\\
&\ldots\\
\overline{\mathbf{cond}(pr_1, pr_2, pr_3)} &= \mathbf{pair}(\mathbf{str}(\mathsf{cond}), \mathbf{pair}(\overline{pr_1}, \mathbf{pair}(\overline{pr_2}, \overline{pr_3})))\\
&\ldots
\end{aligned}$$

We can also test whether such a tree $\overline{pr}$ represents an element of **CP** or **Val**.

Interpreters Written in Our Language

It is possible to write a function in our programming language that acts as an interpreter.

- It takes in a value $\overline{pr}$, representing a closed program pr.
- It begins evaluating pr, using the representation $\overline{pr}$.
- If this evaluation results in an error, then the interpreter returns $\overline{\textbf{const}(\text{nil})}$.
- Otherwise, if it results in a value $\overline{pr'}$ representing a value pr', then it returns $\overline{pr'}$.
- Otherwise, it runs forever.

E.g., $\overline{\textbf{cond}(\textbf{const}(\text{true}), \textbf{const}(\text{false}), \textbf{const}(\text{nil}))}$ evaluates to $\overline{\textbf{const}(\text{false})}$.

Interpreters

We can also write a function in our programming language that acts as an *incremental* interpreter.

- At each stage of its evaluation of a closed program, it carries out some fixed number of steps of the evaluation.
- If during the execution of those steps, an error is detected, then it returns **const**(nil).
- Otherwise, if a value $\overline{pr'}$ representing a value pr' has been produced, then it returns this value.
- But otherwise, it returns an anonymous function that when called will continue this process.

Total Programs and the Meaning of Programs

A *string predicate program* pr is a closed program such that, for all strings w,

$$\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) \in \{\mathbf{norm}(\mathbf{const}(\mathsf{true})), \mathbf{norm}(\mathbf{const}(\mathsf{false}))\}.$$

A string w is *accepted by* a closed program pr iff $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$.

We write $L(pr)$ for the set of all strings accepted by a closed program pr. When this set is a language, then we refer to $L(pr)$ as the *language accepted by* pr.

(E.g., if $pr = \mathbf{lam}(\mathsf{x}, \mathbf{const}(\mathsf{true}))$, then $L(pr) = \mathbf{Str}$, and so is not a language.)

Partially Checking for Acceptance in Forlan

The `Prog` module also includes:

```
val accepted : prog -> str * int -> unit
```

Let's put in the file `5.1-zeros-ones-twos-prog` the following string predicate program, which tests whether a string is an element of $\{\, 0^n 1^n 2^n \mid n \in \mathbb{N} \,\}$.

```
lam
(x,
 letSimp(equal,
         lam(p,
             calc(isZero,
                  calc(compare, var(p)))),
 letSimp(succ,
         lam(n, calc(plus, pair(var(n), int(1))))),
```

Checking for Acceptance in Forlan

```
letSimp
(count,
 lam(a,
     letRec(f, xs,
            cond(calc(isNil, var(xs)),
                 pair(int(0), const(nil)),
                 cond(app(var(equal),
                          pair(calc(fst, var(xs)),
                               var(a))),
                      letSimp(res,
                              app(var(f),
                                  calc(snd, var(xs))),
                              pair(app(var(succ),
                                       calc(fst,
                                            var(res))),
                                   calc(snd, var(res)))),
                      pair(int(0), var(xs)))),
            var(f))),
```

Checking for Acceptance in Forlan

```
letSimp
(xs,
 calc(strToSymList, var(x)),
letSimp
(zeros,
 app(app(var(count), sym(0)), var(xs)),
letSimp
(ones,
 app(app(var(count), sym(1)), calc(snd, var(zeros))),
```

Checking for Acceptance in Forlan

```
letSimp
(twos,
 app(app(var(count), sym(2)), calc(snd, var(ones))),
     cond(calc(isNil, calc(snd, var(twos))),
          cond(app(var(equal),
                   pair(calc(fst, var(zeros)),
                        calc(fst, var(ones)))),
               app(var(equal),
                   pair(calc(fst, var(ones)),
                        calc(fst, var(twos)))),
               const(false)),
          const(false))))))))))
```

Checking for Acceptance in Forlan

```
- val pr = Prog.input "5.1-zeros-ones-twos-prog";
val pr = - : prog
- val test = Prog.accepted pr;
val test = fn : str * int -> unit
- test(Str.fromString "000111222", 100);
unknown if accepted or rejected
val it = () : unit
- test(Str.fromString "000111222", 1000);
accepted
val it = () : unit
- test(Str.fromString "00011222", 1000);
rejected with false
val it = () : unit
- test(Str.fromString "0001112223", 1000);
rejected with false
val it = () : unit
```

Checking for Acceptance in Forlan

```
- val pr = Prog.fromString "const(nil)";
val pr = - : prog
- Prog.accepted pr (Str.fromString "01", 1);
rejected not with false
val it = () : unit
```

Recursive and Recursively Enumerable Languages

We say that a language L is:

- *recursive* iff $L = L(pr)$, for some string predicate program pr; and
- *recursively enumerable* (*r.e.*) iff $L = L(pr)$, for some closed program pr.

We define

$$\mathbf{RecLan} = \{\, L \in \mathbf{Lan} \mid L \text{ is recursive} \,\}, \text{ and}$$
$$\mathbf{RELan} = \{\, L \in \mathbf{Lan} \mid L \text{ is recursively enumerable} \,\}.$$

Hence $\mathbf{RecLan} \subseteq \mathbf{RELan}$. Because $\mathbf{CP}$ is countably infinite, we have that $\mathbf{RecLan}$ and $\mathbf{RELan}$ are countably infinite, so that $\mathbf{RELan} \subsetneq \mathbf{Lan}$. Later we will see that $\mathbf{RecLan} \subsetneq \mathbf{RELan}$.

Proposition 5.1.4

*For all $L \in$ **Lan**, L is recursive iff there is a closed program pr such that, for all $w \in$ **Str**:*

- *if $w \in L$, then* $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\text{true}))$*; and*
- *if $w \notin L$, then* $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\text{false}))$.

Proof. ("only if") Since L is recursive, $L = L(pr)$ for some string predicate program pr. Suppose $w \in$ **Str**. There are two cases to show.

- Suppose $w \in L$. Since $L = L(pr)$, we have that $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\text{true}))$.
- Suppose $w \notin L$. Since $L = L(pr)$, we have that $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) \neq \mathbf{norm}(\mathbf{const}(\text{true}))$. But pr is a string predicate program, and thus $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\text{false}))$.

Proof (cont.). **("if")** To see that pr is a string predicate program, suppose $w \in \mathbf{Str}$. Since $w \in L$ or $w \notin L$, we have that $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) \in \{\mathbf{const}(\text{true}), \mathbf{const}(\text{false})\}$.

We will show that $L = L(pr)$.

- Suppose $w \in L$. Then $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\text{true}))$, so that $w \in L(pr)$.
- Suppose $w \in L(pr)$, so that $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\text{true}))$. If $w \notin L$, then $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\text{false}))$—contradiction. Thus $w \in L$.

□

Proposition 5.1.5
*For all $L \in$ **Lan**, L is recursively enumerable iff there is a closed program pr such that, for all $w \in$ **Str**,*

$$w \in L \text{ iff } \mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\text{true})).$$

Proof. **("only if")** Since L is recursively enumerable, $L = L(pr)$ for some closed program pr. Suppose $w \in$ **Str**.

- Suppose $w \in L$. Since $L = L(pr)$, we have that $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\text{true}))$.
- Suppose $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\text{true}))$. Thus $w \in L(pr) = L$.

Proof (cont.). **("if")** It suffices to show that $L = L(pr)$.

- Suppose $w \in L$. Then $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$, so that $w \in L(pr)$.
- Suppose $w \in L(pr)$. Then $\mathbf{eval}(\mathbf{app}(pr, \mathbf{str}(w))) = \mathbf{norm}(\mathbf{const}(\mathsf{true}))$, so that $w \in L$.

□

Relationship Between the Context-free and R.E. Languages

Theorem 5.1.6
The context-free languages are a proper subset of the recursive languages: **CFLan** $\subsetneq$ **RecLan**.

Proof. To see that every context-free language is recursive, let L be a context-free language. Thus there is a grammar G such that $L = L(G)$. With some work, we can write and prove the correctness of a string predicate program *pr* that implements our algorithm (see Section 4.3) for checking whether a string is generated by a grammar. Thus L is recursive.

To see that not every recursive language is context-free, let $L = \{ 0^n 1^n 2^n \mid n \in \mathbb{N} \}$. In Section 4.10, we learned that L is not context-free. And in the preceding subsection, we wrote a string predicate program *pr* that tests whether a string is in L. Thus L is recursive. $\square$