Experiments with Implementations of two Theoretical Constructions

Torben Amtoft Hansen
Thomas Nikolajsen
Jesper Larsson Träff
Neil D. Jones

DIKU, Department of Computer Science, University of Copenhagen.
Universitetsparken 1, DK-2100 Copenhagen Ø, Denmark
E-mail adresses: amtoft@diku.dk, thomas@diku.dk, traff@diku.dk, neil@diku.dk

Abstract

This paper reports two experiments with implementations of constructions from theoretical computer science. The first one deals with Kleene's and Rogers' second recursion theorems and the second is an implementation of Cook's linear time simulation of two way deterministic pushdown automata (2DPDAs). Both experiments involve the treatment of programs as data objects and their execution by means of interpreters.

For our implementations we have been using a small LISP-like language called Mixwell, originally devised for the partial evaluator MIX used in the second experiment. LISP-like languages are especially suitable since programs are data (S-expressions) so the tedious coding of programs as Gödel numbers so familiar from recursive function theory is completely avoided.

We programmed the constructions in the standard proofs of Kleene's and Rogers' recursion theorems and found (as expected) the programs so constructed to be far too inefficient for practical use. We then designed and implemented a new programming language called Reflect in which Kleene and Rogers "fixed-point" programs can be expressed elegantly and much more efficiently. We have programmed some examples in Reflect in an as yet incomplete attempt to find out for which sort of problems the second recursion theorems are useful program generating tools.

The second experiment concerns an automaton that can solve many non-trivial pattern matching problems. Cook [4] has shown that any 2DPDA can be simulated in linear time by a clever memoization technique. We wrote a simple interpreter to execute 2DPDA programs and an interpreter using Cook's algorithm, and we observed that the latter was indeed much faster on certain language recognition problems. Both have, however, a high computational overhead, since they in effect work by interpretation rather than compilation. In order to alleviate this we applied the principle of partial evaluation, see [5], to specialize each of the two interpreters to fixed 2DPDAs. The result was a substantial speedup.

1 The Second Recursion Theorems

This section deals with a practical and, as it turns out, reasonably efficient implementation of the so-called second recursion theorems of Kleene, [9], and Rogers, [11]. For a much more thorough but also more theoretical treatment, see these or any other good textbook on the subject, e.g. [10].

We give a brief review of the fundamentals, then discuss our implementation of the theorem(s) and report a few experiments devised to demonstrate the adequacy of the implementation and reveal some of the power hidden in the recursion theorems.

Let D be a set of *texts*, programs as well as data objects, closed under formation of pairs. That is, for elements $x, y \in D$ we can form their pair (x, y), which is also in D, and each pair in D can be uniquely decomposed into its constituents. Tuples can be formed by repeated pairing: we adopt the convention that $(x_1, x_2, \ldots, x_n)$ is just $(x_1, (x_2, (\ldots, x_n)\ldots))$. A good example of such a set is the set of LISP S-expressions. A *semantic function*, $\phi : D \to (D \to D)$ that takes programs $p \in D$ and maps them into *computable partial functions* $\phi(p) : D \to D$ is called a *programming language*. In accordance with standard notation in recursion theory, $\phi(p)$ is written φ_p; the n-ary function $\varphi_p^n(x_1, \ldots, x_n)$ is defined to be $\lambda(x_1, \ldots, x_n).\varphi_p((x1, \ldots, x_n))$. The superscript will generally be omitted.

The following definition, originating from [11], captures properties sufficient for a development of (most of) the theory of computability independent of any particular model of computation.

Definition 1.1 *The programming language ϕ is said to be an* acceptable programming system *if it has the following properties:*

1. Completeness property*: for any effectively computable function, $\psi : D \to D$ there exists a program $p \in D$ such that $\varphi_p = \psi$.*

2. Universal function property*: there is a* universal function program *$up \in D$ such that for any program $p \in D$, $\varphi_{up}(p, x) = \varphi_p(x)$ for all $x \in D$.*

3. s-m-n function property*: for any natural numbers m, n there exists a program $sp_n^m \in D$ such that for any program $p \in D$ and any input $(x_1, \ldots, x_m, y_1, \ldots, y_n) \in D$*

$$\varphi_p^{m+n}(x_1, \ldots, x_m, y_1, \ldots, y_n) = \varphi^n_{\varphi^{m+1}_{sp_n^m}(p, x_1, \ldots, x_m)}(y_1, \ldots, y_n)$$

$\varphi^{m+1}_{sp_n^m} : D^{m+1} \to D$ is called the s-m-n function*, written s_n^m.*

The properties listed above actually correspond to quite familar programming concepts. The completeness property states that the language is as strong as any other computing formalism. The universal function property amounts to the existence of a *self-* or *meta-circular* interpreter of the language, whereas the s-m-n function property ensures the possibility of *partial evaluation*: a program $p \in D$ can, when given some of its input, $x \in D$ be specialised to a *residual program*, $p' = s_1^1(p, x)$ which when given the remaining input, y, yields the same result as p applied to all of the input, $\varphi_p(x, y) = \varphi_{s_1^1(p,x)}(y)$. The function s_1^1 is represented by a program in the language, $sp_1^1 \in D$, i.e. $s_1^1 = \varphi_{sp_1^1}$. This concept seems to become increasingly more important, since it has been shown that for instance compilation is a special case of partial evaluation; exploiting the power of the seemingly innocent s-m-n property, however, depends on the existence of non-trivial *partial evaluators*. Such a program is used in the next section. The topic is investigated much further in [7], see also [5]. One observes that by letting *programs and data* have the same form, the theory can be developed without the trick of Gödel numbering. This is a substantial advantage.

The LISP-like language Mixwell, in which programs and data are S-expressions, has been the basis for the implementations. A concrete Mixwell program corresponds to a system of recursive equations and the language is easily (in [2]) proven to have all three of the properties above:

Lemma 1.2 Mixwell *is an acceptable programming system.*

1.1 The theorems

We are now in position to state and sketch the proofs of the second recursion theorems.

Theorem 1.3 (The second recursion theorem, Kleene's version (1938)) *For any program $p \in D$ there is a program $e \in D$ such that $\varphi_p(e, x) = \varphi_e(x)$. We call such an e a* Kleene fixed-point *for p.*

PROOF: By the s-m-n property there is an effectively computable function $s : D \to D$ such that for any program, $p \in D$

$$\varphi_p(y, x) = \varphi_{s(p,y)}(x)$$

namely $s = s^1_1$. Now, it is evidently possible to construct a program $q \in D$ such that

$$\varphi_q(y, x) = \varphi_p(s(y, y), x)$$

Let e be the program $s(q, q)$. Then we have

$$\varphi_p(e, x) = \varphi_p(s(q, q), x) = \varphi_q(q, x) = \varphi_{s(q,q)}(x) = \varphi_e(x)$$

□

Theorem 1.4 (The second recursion theorem, Rogers' version (1967)) *Any computable function, f, taking programs as input (a* program transformation*) has a fixed-point program, that is a program n such that the function computed by n is the same as that computed by the transformed program $f(n)$. Formally, there is an $n \in D$ such that*

$$\varphi_n(x) = \varphi_{f(n)}(x)$$

whenever $f(n)$ is defined. For a program p with $\varphi_p = f$, this n is called a Rogers fixed-point *for p.*

The direct proof of Rogers' version of the recursion theorem is a bit more involved than that of Kleene's, see [11]. Due to the following propositions, the two apparently different theorems are of equal power in the sense that given the ability to find Kleene fixed-points Rogers fixed-points can be found as well and vice versa. By *the* second recursion theorem is therefore meant the version most convenient for the purpose at hand.

Proposition 1.5 *Theorem 1.4 and the s-m-n property implies theorem 1.3.*

Proposition 1.6 *Theorem 1.3 together with the universal function property implies theorem 1.4.*

PROOF: Let f be any program transformation with $f = \varphi_{fp}$. Due to the universal function property we can define a program gp, such that

$$\varphi_{gp}(p, x) = \varphi_{up}(\varphi_{fp}(p), x) = \varphi_{\varphi_{fp}(p)}(x) = \varphi_{f(p)}(x)$$

when $f(p)$ is defined. By theorem 1.3 the program gp has a fixed-point, that is, there is an e with $\varphi_e(x) = \varphi_{gp}(e, x)$. Now

$$\varphi_e(x) = \varphi_{gp}(e, x) = \varphi_{f(e)}(x)$$

So e is a Rogers fixed-point for the transformation f. □

In the proofs of theorem 1.3 and proposition 1.6 fixed-points are obtained in a uniform manner. The second recursion theorem can therefore be generalized a bit:

Proposition 1.7 *There are total computable functions,* $\mathrm{kfix}, \mathrm{rfix} : D \to D$*, such that for any program $p \in D$,* $\mathrm{kfix}(p), \mathrm{rfix}(p)$ *are Kleene respectively Rogers fixed-points for p.*

We leave out a detailed discussion of the relation between the second recursion theorem and the presumably better known first recursion theorem. The first recursion theorem yields *least fixed-points* for *extensional* program transformations (computable functions, f, satisfying $\varphi_{f(n)} = \varphi_{f(m)}$ whenever $\varphi_m = \varphi_n$), whereas the second recursion theorem holds for non-extensional cases as well. A fixed-point obtained by the second recursion theorem is not necessarily least. However Rogers, [11], has shown that from an extensional program transformation, f, a transformation, g, can be constructed such that $\varphi_{f(n)} = \varphi_{g(n)}$ and such that the Rogers fixed-point for g is least. It seems fair to say then that the second recursion theorem is strictly more powerful than the first. This fact has stimulated the interest in "fixed-point programming".