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".

1.2 Some applications

Example 1: self-reproduction

Let $r \in D$ be a program with $\varphi_r(p,x) = p$. According to theorem 1.3, there is a program e, such that for all x

$$\varphi_e(x) = \varphi_r(e,x)$$

and hence

$$\varphi_e(x) = e$$

Now, e is a program that outputs its own text regardless of its input.

Example 2: eliminating recursion

Consider the computable function

$$f = \lambda(p,x).[x = 0 \rightarrow 1, x * \varphi_{up}(p, x-1)]$$

By theorem 1.3, f has a fixed-point, e, with the property

$$\varphi_e(x) = f(e,x) = [x = 0 \rightarrow 1, x * \varphi_{up}(e, x-1)] = [x = 0 \rightarrow 1, x * \varphi_e(x-1)]$$

Thus, φ_e, which was found without explicit use of recursion, is the factorial function. It follows that any acceptable programming system is "closed under recursion".

Until now these are more or less the only examples of fixed-point programming solving computationally interesting problems.

Example 3: unsolvability results

The second recursion theorem can be used to give a very elegant proof (see [10] or [11]) of

Theorem 1.8 (Rice's theorem) *Let P be the set of partial recursive functions and let C be a proper, nonempty subset of P. Then it is undecidable whether, for a given program $p \in D$, φ_p belongs to C or not.*

The theorem is important (although negative in content) stating that it is not possible in general to deduce a program's functional properties from its text.

Example 4: abstract complexity theory

The second recursion theorem has interesting (alas, also negative) applications in so-called abstract complexity theory, cf. Blum, [3]. In the following ψ is an *abstract resource measure*, $\psi_p(n)$ giving the resources used when running program p on input n, satisfying natural requirements stated in [3]. We state without proof:

Theorem 1.9 (The Speed-up theorem) *Let r be a total recursive function of 2 variables. Then there exists a total recursive function f taking values 0 and 1 with the property that to every program p computing f there exists another program q computing f such that*

$$\psi_p(n) > r(n, \psi_q(n))$$

for almost all n.

The theorem is due to Blum, [3]. Given r, f can be found constructively, but going from a program p for f to the "faster" program q is a non-constructive operation. The theorem is therefore of limited interest for practitioners.

1.3 Implementing the recursion theorems

Following the recipe implied by the proof of theorem 1.3, it is very easy to give a straight-forward implementation of Kleene's version of the second recursion theorem. In **Mixwell** we have written a program that when given another program as input, computes its Kleene fixed-point thus realizing proposition 1.7 on the computer. This procedure has some drawbacks. First, the fixed-point programs are rather large, since they contain both the whole input program text plus code (s-m-n function etc.) allowing the fixed-point program to reference its own text. Second, since applications will often involve interpretation (applying the universal function which then must be present in the fixed-point program), a direct implementation easily becomes inefficient, since every nested call to the universal function results in an extra level of interpretation. For example running a fixed-point program to compute the factorial of n results in n levels of interpretation, each one slowing down execution by a large constant factor. Lesson: A too direct implementation is not practical.

1.4 The Reflect language

To overcome these difficulties a new language has been devised containing the essential features for expressing fixed-points directly. This language, **Reflect**, is just **Mixwell** extended with two new features:

1. A built-in universal function (`univ` p i) that takes as arguments a program $p \in D$ and some input $i \in D$, and returns $\varphi_p(i)$. Each (`univ` ...) expression gives rise to a call of the given, fixed **Reflect** interpreter and thus adds no extra levels of interpretation.

2. A special expression *, the value of which is the whole text of the program being executed.

It can be proven that **Reflect** indeed has the power to express fixed-points, done in theorem 1.11 below. The proof relies on a near-denotational semantics of the language, which is an almost trivial extension of the truly denotational semantics of **Mixwell**. Only a sketch is given here.

The semantic functions are

$$\mathcal{E}_R : Exp \to Env \to Prog \to Value$$

$$\mathcal{I}_R : Prog \to Value \to Value$$

where $Exp, Prog, Env, Value \subseteq D$ are the syntactic and semantic domains. $\mathcal{E}_R$ evaluates an expression in an environment. The program, $p \in Prog$, is necessary to give access to functions (not bound in the environment) in this semantics. It also gives access to the program text as a whole. In particular, for the special expression * we have

$$\mathcal{E}_R[\![*]\!]rp = p$$

$\mathcal{I}_R$ executes a whole program given its input. Hence the semantic function of **Reflect** is $\varphi_p = \mathcal{I}_R[\![p]\!]$.

It is an easy exercise to prove

Lemma 1.10 **Reflect** *is an acceptable programming system.*

Theorem 1.11 *Given a* **Reflect** *program p with the following structure:*

```
((p1 (f x) = body1)
 (p2 (...) = ... )
  ...
```

The following Reflect *program e is a Kleene fixed-point for p satisfying* $\varphi_p(e,x) = \varphi_e(x)$.

```
((fix (x) = (univ '((p1 (f x) = body1)
                    (p2 (...) = ... )
                     ... )
                   (list * x) ))
)
```

PROOF: Let x be an input, and let r be the environment defined by $r = ((x.v))$. Now

$$\varphi_e(v) = \mathcal{I}_R[\![e]\!](v) = \mathcal{I}_R[\![w_1]\!]w_2$$

where

$$w_1 = \mathcal{E}_R[\![\texttt{'((p1 ...)...)}]\!]re$$

and

$$w_2 = \mathcal{E}_R[\![\texttt{(list * x)}]\!]re$$

A quoted text evaluates to itself, the `(list ...)` expression to a tuple, and so we get

$$\mathcal{I}_R[\![w_1]\!]w_2 = \mathcal{I}_R[\![p]\!](e,v) = \varphi_p(e,v)$$

The desired equation holds so e is a Kleene fixed-point for the program p. □

1.5 Experiments with the Reflect system

Example 1

The very essence of the second recursion theorem is the ability for a program to reference its own text. Consequentially, self-reproducing programs are very easily expressed in Reflect:

```
((selfreproduce (n) = *))
```

Running the program produces exactly the text shown above.

Example 2

Let f be a program transformation with $\varphi_{f(p)}(x) = p$. The Rogers' fixed-point, n, of such an f is a self-reproducing program, since $\varphi_n(x) = \varphi_{f(n)}(x) = n$. Following the construction used in proposition 1.6 (with some trivial simplifications), this fixed-point can also be expressed as a Reflect-program.

```
((rogers (n) = (call g * n))
 (g (p x) = (univ (call f p) (list x)))
 (f (n) = (list (list 'selfreproduce (list 'x) '= (list 'quote n)))) )
```

Running this program results in the text just given.

Example 3

Elimination of recursion is also possible in Reflect:

```
((fact (n) = (if (n = '1)
                 then '1 else (times n (univ * (list (difference n '1)))))) )
```

The time to compute n factorial grows linearly in n, so a major improvement has occurred over the direct implementation.

Example 4

Suppose mutual recursive functions are not permitted in the programming language at hand. This can be compensated for by employing an intensional program transformation: each of the forbidden calls are replaced by a call to the universal function with a transformed program in which the function to be called appears as the first one (in **Reflect** the meaning of a program is determined by its first function). A simple example is shown below:

```
((even (x) = (if (null x) then 'true else
                 (univ (list (car (cdr *)) (car *)) (list (cdr x))) ))
 (odd (x) =  (if (null x) then 'false else
                 (univ (list (car (cdr *)) (car *)) (list (cdr x))) )) )
```

Example 5

Memoization seems to be a natural application of self-modification. Whenever some value from a function is returned a new program is built with this value incorporated into it and further applications of the function are done in this new context.

A memoizing version of the "map-functional", applying a function f to a list of arguments is given below, with the fibonacci function as f. Function **seen** checks whether the function has already been evaluated on the current argument; if not, a new program is constructed by extending the definition of **seen** thus building $f(x)$ into it.

```
((map (xs) = (if (null xs) then 'nil else
             (call maptest (call seen (car xs)) (car xs) (cdr xs))) )

 (maptest (xval x xs) = (if (equal xval 'newval)
                          then (call mapmem (call fib x) x xs) ; new value
                          else (xval :: (call map xs))) )      ; f(x) seen

 (mapmem (xval x xs) = (xval :: (univ (call newmap x xval) (list xs))) )

; build a new program by extending the body of the 'seen' function.
 (newmap (x xval) =
         (let (eqn1 eqn2 eqn3 eqn4 (_ _ _ body) fct) = * in
              (let newseen =
                   (list 'seen (list 'x) '=
                         (list 'if (list 'equal 'x (list 'quote x))
                               (list 'quote xval)
                               body))
                   in (list eqn1 eqn2 eqn3 eqn4 newseen fct))) )

 (seen (x) = 'newval)  ; initially nothing has been seen.

; function to be mapped and memoized. The recursive call has been replaced
; with a call to map to take advantage of already computed values.
 (fib (x) = (if (x = '0) then '0 elsf (x = '1) then '1 else
             (callx plus (car (call map (list (callx difference x '2))))
                         (car (call map (list (callx difference x '1))))) )) )
```

The effect of such memoization can be considerable. Given the list of integers up to n in ascending order the fibonacci numbers are computed in near linear time.

1.6 Remarks and open questions

The language **Reflect** is a good starting point for experiments in fixed point programming. It is a low-level language and lacks convenient tools for specifying program transformations, but such problems can to a large extent be solved by adding syntactic sugar and more built-in functions. More important: fixed-point programs can be expressed naturally and executed with reasonable efficiency, giving a framework for experiments with constructive applications of the second recursion theorem. The most pressing problem is to widen the spectrum of computationally interesting applications.

2 2DPDAs and Cook's Theorem

2.1 Introducing 2DPDAs

A two-way deterministic pushdown automaton (2DPDA) is a Turing machine-like device with a finite state control and a two-way input tape with symbols from a finite alphabet. In contrast to the Turing machine a 2DPDA also has an unbounded pushdown stack that can hold symbols from another finite alphabet, and the tape is read-only and of finite length.

2.2 Cook's theorem

2DPDAs are interesting because they are powerful enough to solve many non-trivial pattern matching problems [1]; yet they can be simulated in time bounded by a linear function of the input length on a uniform Random Access Machine (RAM), [4]. See also [1]. This is:

Theorem 2.1 (Cook) *Every 2DPDA can be simulated in time O(n), where n is the length of the input tape, on a uniform RAM.*

2.3 Comparing 2DPDAs to other automata

The class of languages generated by 2DPDAs is denoted *2DPDA*. Comparing this with well known classes of languages gives the chain ($\subset$ means strict inclusion):

$$REG \subset DCF \subset 2DPDA \subset RE$$

It is unknown whether $CF \subset 2DPDA$, but the converse is not the case as $\{a^n b^n c^n | n \geq 0\}$ is in *2DPDA* but not in *CF*. It follows that 2DPDAs are rather strong, since they can recognize all LR(k) languages and even some languages that are not context-free.

2.4 The 2DPDA programming language

To use 2DPDAs we have chosen to formulate them as programs in a primitive imperative programming language (labels, guarded commands, push and pop instructions, goto's and tape movement commands). The language, also named 2DPDA, will not be described formally here, as we do not have space in this summary and the semantics is straightforward.

Definition 2.2 *Given an input tape* $\triangleright a_1 a_2 \ldots a_n \triangleleft$, *an* instantaneous description, ID *for short, of a given 2DPDA program P, is a tuple:*

$$D = (l, p, s_1 \ldots s_m) = (label, input\ tape\ read\ head\ position, stack\ contents)$$

where l is a label in P, $p \in \{0, \ldots, n+1\}$ *and* $s_1 \ldots s_m$ *is a string of symbols from the stack alphabet. The symbols* $\triangleright$ *and* $\triangleleft$ *are endmarkers not in the tape alphabet.*

An ID is sufficient to represent a state of computation of a 2DPDA.

The semantics for a 2DPDA program is defined by a *transition function* $\delta : ID \rightarrow ID$, where ID is the set of IDs for the program. As an example of a 2DPDA program we give a recognizer for $\{xcy | x, y \in \{a, b\}^* and\ y\ is\ a\ substring\ of\ x^R\}$.

```
(
(copy-x  ((iftape= c) right (goto compare))                ;copy x onto stack.
         ((iftape= a) (push a) right (goto copy-x))        ;
         ((iftape= b) (push b) right (goto copy-x))        ;
)
(compare (rightend? accept)                                ;y did match.
         (bottom? reject)                                  ;y didn't match.
         (iftop= a) (iftape= a) pop right (goto compare))  ;both are a's.
         ((iftop= b) (iftape= b) pop right (goto compare)) ;both are b's.
         (left (goto restore))                             ;y isn't prefix of x.
)
(restore ((iftape= c) pop right (goto compare))            ;stack restored; pop.
         ((iftape= a) (push a) left (goto restore))        ;restore a.
         ((iftape= b) (push b) left (goto restore))))      ;restore b.
```

The program just tries to match y against a prefix of x, and if it fails it removes the first character of x and repeats until the match succeeds or the suffix of x becomes shorter than y. The algorithm takes $O(|x||y|)$ steps.

2.5 Cook's Construction

We will now describe a 2DPDA simulator which runs in time $O(n)$, where n is the length of the input tape. The idea is due to Cook [4] and the algorithm we use is a near applicative version of the one given in [6]. We will use *surface configurations* as a weak form of IDs, where only the stack top is given:

Definition 2.3 *Given an input tape* $\triangleright a_1 a_2 \ldots a_n \triangleleft$, *a* surface configuration C *(or just configuration) of a 2DPDA program* P *is a tuple:*

$$C = (l, p, s) = (label, input\ tape\ read\ head\ position, stack\ top\ symbol)$$

where l *is a label in* P, $p \in \{0, \ldots, n+1\}$ *and* s *is a symbol from the stack alphabet or "bottom".*

A surface configuration C of a program P is essentially an ID of P with a one element stack, and the transition function δ for P is then defined on C. If a configuration C is given, then we *can* make some transitions: all P's actions until it needs to know what is below the given stack top. The next two definitions formalize this.

Definition 2.4 *A configuration* C *of a given 2DPDA program* P *is said to be* terminal *iff* δ, *the transition function for* P, *used on* C *will either make* P *stop or it gives an ID* (l, p, s_0), *where* s_0 *is the empty stack.*

Definition 2.5 *A configuration* C *of a given 2DPDA program* P *is said to be a* terminator *of a configuration* C' *of* P *iff* C *is terminal and there is a sequence* $D_1, \ldots, D_k$ *of ID's of* P, *none of them with empty stack, which fulfills:*

$$\delta(C') = D_1,\ \delta(D_i) = D_{i+1}\ for\ all\ i\ \in \{1 \ldots k-1\},\ \delta(D_k) = C$$

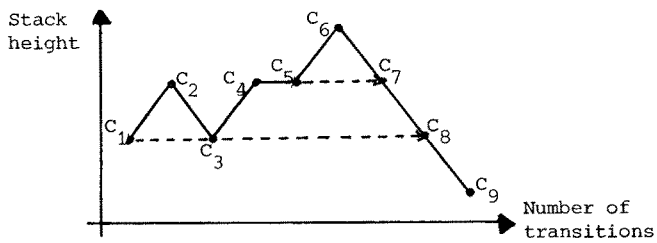

Figure 1: A mountain, illustrating the use of configurations.

Note that a terminator is unique, if it exists, because a 2DPDA is deterministic. To illustrate the notion of a terminator we give a "mountain" in figure 1. Here C_2, C_6, C_7, C_8 are terminal configurations. C_8 is the terminator for C_1 and C_3, and C_7 is the terminator for C_4 and C_5.

How many steps can we execute when we do not know what is below a given one element stack? For a given configuration C, we just have to follow the x axis (δ transitions) from C until the mountain reaches a lower altitude than that of C, this being the first step that we will not be capable of doing.

The technique used in [6] is to simulate P's actions, meanwhile storing all pairs (conf,term) of a configuration and its terminator that are encountered during the interpretation. If P enters "conf" again, this stored information is used to take a short cut, avoiding the need to redo all the single transition steps from "conf" to "term". In section 2.7 it is proven that this technique yields a linear time algorithm.

If the 2DPDA we are simulating terminates, then the idea given above will work because every reachable configuration has a terminator. If it does not terminate the above idea can still be used after some elaboration.

2.6 The Cook simulator

We now present our memoizing interpreter in pseudo Mixwell code. For that purpose we define the *followterminator* ("followterm") of a configuration C as the configuration following the terminator of C, with "bottom" as a (dummy) stack symbol (the actual stack symbol will be filled in from context).

Some remarks:

1. The `table` is a structure where already computed followterms can be found.

2. The main function `Simulate` returns for a given configuration a pair, the first component of which is the followterm of the configuration (or the answer `accept` or `reject`). The second component is an updated table.

3. The `c-list` is a list of configurations, which will have the same followterm as the current configuration. When this followterm is found, it will thus be possible to insert all those configurations into `table`. This is done by `Update`.

4. `Next-conf` performs the transition function on a given configuration, i. e. interprets one step of the 2DPDA and returns the resulting configuration (or the answer 'accept or 'reject).

```
Cook (aut input) =
 let init-conf = (Init-conf aut input) in
  let (answer . new-table) = (Simulate init-conf 'nil init-table aut)
   in answer
```

```
Simulate (conf c-list table aut) =
  if the followterminator of conf "followterm" can be found in table
              (by function Lookup)
  then (followterm :: (Update table c-list followterm))
  else let new-conf   = (Nextconf conf aut)
           new-c-list = (conf :: c-list) in
   case new-conf is
    'accept: ('accept :: table)
    'reject: ('reject :: table)
    the result of a pop: (new-conf :: (Update table new-c-list new-conf))
    the result of a push:
         let (followterm . table1) = (Simulate new-conf 'nil table aut) in
          if followterm = 'accept then ('accept :: table1) else
           if followterm = 'reject then ('reject :: table1) else
            let followterm1 = (followterm with the stacksymbol of conf inserted) in
             (Simulate followterm1 new-c-list table1 aut)
    the result of no stack operations:
         (Simulate new-conf new-c-list table aut)
```

2.7 Time analysis of the simulation algorithm

We shall now analyze the behaviour of the simulation on a terminating 2DPDA program and prove it to be linear with respect to input size, provided that table access and update can be done in constant time. The proof is not too different from the one given in [6]. However, this version does not rely on the input head only moving one position at a time.

We assume that during an activation of `Simulate` with some configuration C as parameter, a new call to `Simulate` with C as parameter is not made (if that happened the program would be in an infinite loop). We will start by showing an essential proposition:

Proposition 2.6 *Given some configuration C. Then C will be placed in a* `c-list` *at most once during program execution.*

PROOF: Follows from the fact that configurations with already computed followterms are not inserted in a `c-list`, together with our assumption about no loops. □
We now obtain:

Proposition 2.7 *During program execution, the number of successful calls to* `Lookup` *will not exceed the number of reachable configurations.*

PROOF: A successful call to `Lookup` amounts to popping the stack, since we make a short cut to the followterm of the current configuration. But every time we simulate pushing the stack, we insert the previous configuration in a `c-list`. According to proposition 2.6, this only happens once for each reachable configuration; so our assertion follows from the fact that the number of pop operations cannot exceed the number of push operations. □

Proposition 2.8 `Simulate` *is at most activated twice the number of reachable configurations.*

PROOF: When `Simulate` is activated, either a successful call to `Lookup` is made or a configuration is inserted in a `c-list`. According to proposition 2.6 and 2.7, the number of occurrences of each

case is bounded by the number of reachable configurations. □

From the above three propositions, it is easy to see that the total time spent is $O(RC)$, where RC is the number of reachable configurations (split the total time spent in `Lookup` operations, `Update` operations and other operations). As the latter is bounded by a linear function of the input size, we have proven:

Theorem 2.9 *Assuming that table access and update can be made in constant time, and assuming that the program does not loop, Cook's algorithm can be executed in linear time with respect to input size.*

The proof of this theorem furthermore indicates that the number of calls to `Simulate` is a good measure for the actual time consumption of some fixed program.

The halting problem is solvable for 2DPDAs, since a 2DPDA loops on given input if and only if some configuration C is inserted into a `c-list` more than once (if C is inserted twice into the same `c-list`, the loop is with constant stack, if C is inserted into two different `c-lists`, the loop is with growing stack). The simulator could thus be extended to recognize looping at the cost of some extra overhead.

2.8 Experiments with Cook's simulator

We now compare the running time of the memoizing simulator (named `Cook`) with a simple non-memoizing simulator (named `2DPDA`). As measures we will use both the cpu time in milliseconds and the number of calls to the main function `Simulate`. The drawback of using cpu time is the inefficiency of table accesses and updates, which are not easily performed in constant time in our purely functional programming language, and which therefore might prevent `Cook` from being truly linear with respect to input size. On the other hand, since the time required to execute one `Simulate`-loop in `Cook` is clearly longer than the time to execute one step in `2DPDA` (due to the overhead caused by the table manipulation), it seems fair to include cpu time consumption as well.

Our example problem in its "natural" formulation requires exponential time: given a fixed DFA (deterministic finite automaton) and some n, we want to know whether the DFA accepts at least one string of length n. From the DFA, we can construct a 2DPDA to solve the above problem by exhaustive search. If the DFA alphabet has q symbols, the execution time will be $O(q^n)$, where n is the input length.

The idea is to identify a string of n symbols with a number in the q-ary system in the obvious way, the first symbol being the most significant digit. We start by checking whether the string corresponding to 0 is accepted by the DFA (by simulating the state transitions of the DFA). Then check the strings corresponding to $1, 2, \ldots, q^n - 1$.

The main invariant of the 2DPDA algorithm is that the stack at any moment has the content

$$s_{i_0} a_{i_0} s_{i_1} a_{i_1} s_{i_2} a_{i_2} \ldots$$

where $s_{i_{j+1}}$ for any j is the result of applying the DFA's state transition function to (s_{i_j}, a_{i_j}), and where s_{i_0} is the initial state of the DFA. This makes it easy to perform the required "addition of 1".

As a concrete example of a DFA we will use one which given a string in $\{0,1\}^n$ decides whether the string contains an odd number of 0's and an odd number of 1's. The DFA has 4 states: s_0 (corresponding to an even number of 0's and an even number of 1's), s_1 (even 0's and odd 1's), s_2 (odd 0's and even 1's), and s_3 (odd 0's and odd 1's). The derived 2DPDA is listed below:

```
(
(init         ((push s0) (goto push0-s0)) )
(push0-s0     (rightend? (goto next))       ((push 0) right (goto pushnext-s2)) )
(push0-s1     (rightend? (goto next))       ((push 0) right (goto pushnext-s3)) )
(push0-s2     (rightend? (goto next))       ((push 0) right (goto pushnext-s0)) )
(push0-s3     (rightend? accept)            ((push 0) right (goto pushnext-s1)) )
(pushnext-s0 ((push s0) (goto push0-s0)) )
(pushnext-s1 ((push s1) (goto push0-s1)) )
(pushnext-s2 ((push s2) (goto push0-s2)) )
(pushnext-s3 ((push s3) (goto push0-s3)) )
(next         (pop left (goto next-1)) )
(next-1       (leftend? reject)
              ((top= 1) pop (goto next))
              ((top= 0) pop (goto restore-1)) )
(restore-1    ((top= s0) (push 1) right (goto pushnext-s1))
              ((top= s1) (push 1) right (goto pushnext-s0))
              ((top= s2) (push 1) right (goto pushnext-s3))
              ((top= s3) (push 1) right (goto pushnext-s2)) )
)
```

The execution times are listed below. Notice that the results only are interesting for odd n's.

n	2DPDA		Cook	
	loops	*time*	*loops*	*time*
3	67	200	45	220
5	283	780	77	400
7	1147	3260	109	540
9	*memory*	*exhausted*	141	700
11			173	840

As expected, Cook's algorithm is far superior: practically linear with respect to input size, also when we consider cpu time. On the other hand, due to its exponential growth rate it soon becomes impossible to execute 2DPDA.

When interpreting the substring program mentioned previously, such a transformation is not possible, as the program for "normal" inputs is linear already. However, Cook typically saves so many loops during interpretation that it even with respect to cpu time is somewhat faster.

We also have written a program detecting whether a given input string is a palindrome. During the interpretation of this program no use of stored information can ever be made, so in that case Cook only introduces some overhead.

2.9 Partial evaluation of the simulators

Although Cook's simulator in many cases is more efficient than its non-memoizing counterpart, it still suffers from the drawbacks of being an interpreter, e. g. a lot of time is spent in parsing operations. A solution to this problem is to use a *partial evaluator*, as described in [5] and [7]. Then we can partially evaluate Cook with respect to some fixed 2DPDA and obtain a more efficient target program.

A partial evaluator is an implementation of the s_n^m function mentioned in the first part of this paper. There would be no point in using the immediate implementation of the s_n^m function; however,

we have access to a non-trivial partial evaluator named MIX, which in most cases is able to speed up execution time by a significant constant factor. For further information on MIX, see [7].

After making some minor changes to our two interpreters, we have been able to process them by MIX. The MIX-produced target programs contain, for each pair of a stack top and a program label, corresponding specialized versions of `Simulate` (the main function in the interpreter). Such target programs tend to be large, but nevertheless they are quite efficient. Imagine for example that a line in the 2DPDA program looked like this:

```
(start ((iftop= b) (iftape= c) (push a) (goto middle))
```

Let the version of `Simulate` corresponding to label `start` and stack top `b` be called `Sim-1`, and let the version of `Simulate` corresponding to label `middle` and stack top `a` be called `Sim-2`. Then there will appear a function definition in the target program which looks something like this:

```
Sim-1 (rest-stack tape headposition) =
 if the head points at 'c
 then (Sim-2 (b :: rest-stack) tape headposition)
 else ...
```

The test on the stack top has vanished, and no time is spent on parsing.

We have performed several experiments with the target programs produced by MIX. Typically, we gain a factor 5 (in cpu time): running `Cook` with a 2DPDA p and tape d as input is 5 times slower than running the result of partially evaluating `Cook` with respect to p on d.

On the other hand, we often gain a factor of more than 10 by applying MIX to the `2DPDA` interpreter. For instance, while `Cook` as a rule is faster than `2DPDA` when interpreting the substring program mentioned earlier, the target program of `Cook` is slower than the target program of `2DPDA` (This is of course not true for the exponential time program). This is no wonder, since especially `2DPDA` spends a large fraction of its execution time on parsing its input program, and the parsing time is eliminated by MIX. The time spent on `Cook`'s table operations cannot be reduced by the same degree, so we do not gain so much by partially evaluating `Cook`.

2.10 Remarks and open questions

The aim of this experiment was to investigate the effect of combining Cook's idea and partial evaluation: given a problem, one first writes a 2DPDA program to solve it, perhaps in quadratic or exponential time. By applying Cook's idea the problem can be solved in linear time, and by partial evaluation the constant factor can be greatly reduced. The results obtained so far seem promising, but further experiments on more complicated problems should be made. For that purpose, it is almost mandatory to devise a higher level language for programming 2DPDAs, e. g. with a "while"-loop construct, procedures in some form etc.

Further work could also include partially evaluating the Cook interpreter with a known automaton and partially known input tape. Consider for example the substring program. If the assumed substring is known and the assumed superstring is unknown, there exist some very efficient linear algorithms (by Knuth, Morris, Pratt and others). It would be quite nice if one of these algorithms could be obtained in an automatic way by partial evaluation, and this has actually been done recently at Copenhagen (by Olivier Danvy) but within another framework. With 2DPDA's this line has not been pursued further.

References

[1] Alfred Aho, John Hopcroft, Jeffrey Ullman: The Design and Analysis of Computer Algorithms. Addison-Wesley, 1974.

[2] Torben Amtoft, Thomas Nikolajsen, Jesper Larsson Träff: Implementation of some Theoretical Constructs. DIKU, University of Copenhagen, student project report 1988.

[3] Manuel Blum: A Machine-Independent Theory of the Complexity of Recursive Functions. Journal of the Association for Computing Machinery, vol. 14, no. 2, April 1967, pp. 322-336.

[4] Stephen A. Cook: Linear Time Simulation of Deterministic Two-Way Pushdown Automata. Information Processing 71, pp. 75-80, North-Holland Publishing Company, 1972.

[5] A. P. Ershov: Mixed Computation: Potential applications and problems for study. Theoretical Computer Science 18, pp. 41-67, 1982.

[6] Neil D. Jones: A Note on Linear Time Simulation of Deterministic Two-Way Pushdown Automata. Information Processing Letters, vol. 6, no. 4, 1977.

[7] Neil D. Jones, Peter Sestoft, Harald Søndergaard: MIX: A Self-applicable partial Evaluator for Experiments in Compiler Generation (Revised Version). Lisp and Symbolic Computation, issue 2, fall 1988.

[8] Akira Kanda: Theory of Computation. Lecture notes, DIKU, University of Copenhagen, 1985.

[9] Stephen Cole Kleene: Introduction to Metamathematics. North-Holland, 1952.

[10] A. I. Mal'cev: Algorithms and recursive functions. Wolters-Noordhoff Publishing, 1970.

[11] Hartley Rogers: Theory of Recursive Functions and Effective Computability. McGraw-Hill, 1967.