Contents

Computer Implementation and Applications of Kleene's S-m-n and Recursion Theorems

Neil D. Jones

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

Abstract

This overview paper describes new program constructions which at the same time satisfy Kleene's theorems and are efficient enough for practical use on the computer. The presentation is mathematically oriented, and along the way we mention some unsolved problems whose precise formulations and solutions might be of mathematical interest.

We identify a "programming language" with an acceptable enumeration of the recursive functions, and a program with an index. In computing terms, Kleene's s-m-n theorem says that programs can be *specialized* with respect to partially known arguments, and the second recursion theorem says that programs may without loss of computability be allowed to reference their own texts.

The theorems' classical proofs are constructive and so programmable—but in the case of the s-m-n theorem, the specialized programs are typically a bit slower than the original, and in the case of the recursion theorem, the programs constructed in the standard proofs are extremely inefficient. These results were thus of no computational interest until new methods were recently developed [12, 1], two of which are described here.

An important application: it was realized in [9, 21, 8] that one can, in principle, compile from one programming language **S** into another, **L**, by using the s-m-n theorem. Compiling is done by applying the s-m-n theorem to specialize an interpreter for **S** written in **L** to a fixed **S**-program. Further, compiler generation can be done by self-application: specializing an s-1-1 program to a fixed interpreter. A further use of self-application yields a compiler generator: a program that transforms interpreters into compilers.

1 Introduction

In this overview paper we describe some recent Computer Science research results closely related to two classical theorems from recursive function theory. The first concerns the s-m-n theorem and its now well-established applications to compiling and compiler generation [12]. In Computer Science a program with the s-m-n property is called a *partial evaluator*, perhaps better named a *program specializer*. Given a program and fixed values for some but not all of its inputs, a partial evaluator constructs a new

program which, when applied to the values of the remaining inputs, yields the same result the original program would have given on all its inputs.

In the mid 1970's, Futamura in Japan and Turchin and Ershov in the Soviet Union [9, 21, 8] independently discovered the possibility of using partial evaluation for automatic compilation and for compiler generation as well, starting with a language definition in the form of an interpreter. This understanding was, however, only in principle since specialization as done in the classical proofs of the s-m-n theorem is of no computational interest due to its inefficiency. The first computer realization efficient enough to be of practical interest is reported in [12]; these and several subsequent papers have established the utility and conceptual simplicity of this approach.

We also describe a relatively efficient implementation of Kleene's second recursion theorem [1]. While this theorem has many applications in mathematics, it is not yet clear whether it will be as constructively useful in computer science as the s-m-n theorem has turned out to be.

The Problem of Compiler Generation A compiler is a meaning preserving function from "source" programs (expressed in a programming language suitable for humans) into "target" programs (in a language suitable for computer execution). Correctness is of paramount importance: the compiler-generated code must faithfully realize the intended meaning of the source program being compiled. In practice, however, compiler construction is an arduous task, often involving many man-years of work, and it can be quite difficult to ensure correctness.

Program meanings can be defined precisely via denotational or operational semantics. A language's operational semantics can be given in the form of a program called an *interpreter*. In recursion theoretical terms, an interpreter for programming language **S** written in language **L** is an **L**-program computing a universal function for **S**; and a *compiler* from **S** to **L** is a meaning-preserving function from **S**-programs to **L**-programs (precise definitions appear later).

The compiler correctness problem naturally led to the goal of *semantics-directed compiler generation*: from a formal programming language definition automatically to derive a correct and efficient compiler for the defined language.

Once a correct compiler generator has been developed, every generated compiler will be faithful to the language definition from which it was derived. Such a system completely obviates the need for the difficult intellectual work involved in proving individual compilers correct [10, 16]. Ideally its role in the semantic part of practical compiler construction would be similar to that of YACC and other parser generators for syntax analysis.

Industrial-strength semantics-directed compiler generators do not yet exist, but noteworthy progress has been achieved in the past few years. Several systems based on the lambda calculus have been developed ([17, 18, 22], of which [22] is the fastest that has been used on large language definitions). However, all these systems are large and complex, and correctness proofs would be very hard to accomplish.

Partial Evaluation A partial evaluator deserves its name because, when given a program and incomplete input, it will do part of the evaluation the program would do on complete input. A partial evaluator may be fruitfully thought of as a *program specializer*: given a program and the values of part of its input data, it yields another program which,

given its remaining input, computes the same value the original program gives on all its input. In other words, a partial evaluator is an efficient computer realization of Kleene's s-m-n theorem [14].

Consider, for example, the following exponentiation program, written as a recursion equation:

```
p(n,x) =  if n=0     then 1 else
          if even(n) then p(n/2,x)**2 else x*p(n-1,x)
```

The s-m-n theorem is classically proved in a rather trivial way. For example if `n` = 5, a specialized version of the program above could be a system of two equations:

```
p5(x)  =  p(5,x)
p(n,x) =  if n=0     then 1 else
          if even(n) then p(n/2,x)**2 else x*p(n-1,x)
```

A better specialized exponentiation program for `n` = 5 can be got by unfolding applications of the function `p` and doing the computations involving `n`, yielding the residual program:

```
p5(x)  =  x*(x**2)**2
```

The traditional proofs of Kleene's s-m-n theorem do not take efficiency into account and so yield the equivalent of the first specialized program. Efficiency is very important in applications though, so partial evaluation may be regarded as the quest for efficient implementations of the s-m-n theorem.

Partial evaluation and compiling We will see that compilation can be done by partial evaluation, and that compilers and even a compiler generator can be constructed by self-applying a partial evaluator, using an interpreter as input. These possibilities were foreseen in principle in [9, 21, 8], and first realized in practice as described in [12].

By this approach the partial evaluator and the language definition are the only programs involved in compiler generation. The resulting systems are much smaller than the above-mentioned semantics implementation systems. A side effect is that correctness is much easier to establish, e.g. see [11] for a complete correctness proof of a self-applicable partial evaluator for the untyped lambda calculus.

An interpreter for a programming language **S** usually has two inputs: a "source program" p to be executed, and its input data. Compiling from **S** is done by specializing the given **S**-interpreter with respect to p. The resulting "target program" is an optimized version of the interpreter, specialized so that it is always always applied to the same **S**-program p.

An *efficient* target program is obtained by performing at compile time all the interpreter's actions which depend only on source program p.

Compiler generation is more complex in that it involves *self-application*. One can generate a compiler by using the partial evaluator to specialize itself with respect to an interpreter as fixed input. Further, specializing the partial evaluator with respect to itself as fixed input yields a compiler generator: a program that transforms interpreters into compilers. These mind-boggling but practically useful applications of partial evaluation are explained more fully in section 2.

The Recursion Theorem The constructions in the standard proofs of Kleene's recursion theorem, and Rogers' variant as well, were also programmed and found (as expected) to be far too inefficient for practical use. A new programming language was designed and implemented in which Kleene and Rogers "fixed-point" programs can be expressed elegantly and much more efficiently. Several examples have been programmed in an as yet incomplete attempt to find out for which sort of problems the recursion theorems are useful program generating tools.

Outline

In section 2 we review the definition of partial evaluation and state the "Futamura projections", which show in principle how partial evaluation can be used for compiling and compiler generation. In section 3 a concrete programming language is introduced and a self-interpreter for it is sketched. Section 4 begins with program running times and the computational overhead caused by interpretation. Efficiency is discussed in general terms, and we point out the desirability of a complexity theory for partial evaluation, analogous to but in some important respects different from traditional computational complexity theory. Some open problems are presented that appear susceptible to formulation and solution in such a new complexity theory. Finally, section 5 concerns efficient implementation of the Second Recursion Theorem.

Acknowledgements

This paper owes much to the co-authors of [12] and [1], and to discussions and constructive criticism by the DIKU group, including Lars Ole Andersen, Anders Bondorf, Carsten Gomard and Torben Mogensen.

2 Preliminaries

In both partial evaluation and the recursion theorem programs are treated as data objects, so it is natural to draw both programs and their data a single universal domain D. The traditional usage of natural numbers in recursive function theory is simple, abstract and elegant, but involves a high computational price: all program structures and nonnumeric data must be encoded by means of Gödel numbers, and operations must be done on encoded values.

Thus for practical reasons we diverge from the traditional recursion-theoretic use of the natural numbers. The implementations [12, 1] use a small Lisp-like language—especially suitable since Lisp programs *are* data values so programs and data have the same form. Ths avoids completely the need for the tedious coding of programs as Gödel numbers so familiar from number-based recursive function theory. A substantial advantage is that algorithms for the s-m-n and universal functions become much simpler and more efficient. For a concrete example, see section 3.2.

Our choice of D is thus the set of Lisp "S-expressions", defined as the smallest set satisfying:

- Any "atom" is in D, where an atom is a sequence of one or more letters, digits or characters, excluding (,) and blank.

- If $d_1, \ldots, d_n \in D$ for n ≥ 0, then $(d_1 \ldots d_n) \in D$.

2.1 Programming Languages

A *programming language* ϕ is a function $\phi : D \to (D \to D)$ that associates with each element $p \in D$ a partial function $\phi(p) : D \to D$. In accordance with standard notation in recursion theory, $\phi(p)$ will often be written φ_p. The n-ary function $\varphi_p^n(x_1, \ldots, x_n)$ is defined to be $\lambda(x_1, \ldots, x_n).\varphi_p((x_1 \ldots x_n))$. The superscript n will generally be omitted.

Intuitively, a programming language is identified with its "semantic function" ϕ, so $\phi(p)$ is the partial input-output function denoted by program p. *All* elements p of D are regarded as programs; ill-formed programs can for example be mapped to the everywhere undefined function.

The following definition, originating from [19], captures properties sufficient for a development of (most of) the theory of computability independent of any particular model of computation.

Definition 2.1 *The programming language ϕ is 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 total recursive function $s_n^m : D^{m+1} \to D$ such that for any program $p \in D$ and any input $(x_1, \ldots, x_m, y_1, \ldots, y_n) \in D$*

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

s_n^m is called the s-m-n function.

The properties listed above actually correspond to quite familiar 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 for the language which, when applied to the text of a program and its input, computes the same value that the program computes on the same input.

The s-1-1 property ensures the possibility of partial evaluation. Suppose, for example, that program $p \in D$ takes two inputs. When given its first input $d_1 \in D$, program p can be specialised to yield a so-called residual program $p' = s_1^1(p, d_1)$. When given the remaining input d_2, p' yields the same result as p applied to both inputs. This seemingly innocent property has become increasingly more important. Practical exploitation, however, requires non-trivial partial evaluators.

The s-1-1 definition is essentially the so-called *mix equation* central to our earlier articles. By completeness the function s_1^1 must be computed by some program, and such a program is traditionally called *mix* in the partial evaluation literature. Following are definitions of compilers, interpreters and a version of the *mix* equation.

Notational variations References [12, 11] and others write the semantic function in linear notation (to avoid deeply nested subscripts), associate function application from the left, and use as few parentheses as possible. For example, $(\phi(p))(d)$ would be written as $\phi\, p\, d$. Languages are usually denoted by roman letters, the ones we will use being **L** (the default "meta-" or "implementation" language), **S** (a "source language") to be interpreted or compiled, and **T** (a "target language"), the output of a compiler or partial evaluator.

2.2 Compilers, Interpreters and Partial Evaluators

Definition 2.2 *Let* **L**, **S** *and* **T** *be programming languages.*
Program int is an interpreter for **S** written in **L** *if for all s, d* $\in D$

$$\mathbf{S}\ s\ d = \mathbf{L}\ int\ (s,\ d)$$

Program comp is an **S**-to-**T**-compiler written in **L** *if for all* $s, d \in D$

$$\mathbf{S}\ s\ d = \mathbf{T}\ (\mathbf{L}\ comp\ s)\ d$$

Program mix is an **S**-to-**T**-partial evaluator written in **L** *if for all* $p, d_1, d_2 \in D$

$$\mathbf{L}\ p\ (d_1, d_2)\ = \mathbf{L}\ (\mathbf{L}\ mix\ (p, d_1))\ d_2$$

We will only use one-language partial evaluators, for which $\mathbf{L} = \mathbf{S} = \mathbf{T}$. The universal program *up* is clearly an interpreter for **L** written in **L**; and *mix* is a program to compute s_1^1 from the previous definition.

2.3 The Futamura Projections for Compiling and Compiler Generation

Suppose we are given an interpreter *int* for some language **S**, written in **L**. Letting *source* be an **S**-program, a compiler from **S** to **L** will produce an **L**-program *target* such that $\mathbf{S}(source) : D \to D$ and $\mathbf{L}(target) : D \to D$ are the same input-output function.

The following three equations are the so-called "Futamura projections" [9, 8]. They assert that given a partial evaluator *mix* and an interpreter *int*, one may compile programs, generate compilers and and even generate a compiler generator.

$$\begin{array}{lcll} \mathbf{L}\ mix\ (int,\ source) & = & target & \\ \mathbf{L}\ mix\ (mix,\ int) & = & compiler & \\ \mathbf{L}\ mix\ (mix,\ mix) & = & cogen & \text{a compiler generator} \end{array}$$

Explanation Program *source* from the *interpreted* language **S** has been translated to program *target*. This is expressed in the language **L** in which the interpreter is written (natural since *target* is a specialized version of **L**-program *int*). It is easy to verify that the target program is faithful to its source using the definitions of interpreters, compilers and the mix equation:

$$
\begin{aligned}
\mathit{output} &= \mathbf{S}\ \mathit{source}\ \mathit{input} \\
&= \mathbf{L}\ \mathit{int}\ (\mathit{source},\ \mathit{input}) \\
&= \mathbf{L}\ (\mathbf{L}\ \mathit{mix}\ (\mathit{int},\ \mathit{source}))\ \mathit{input} \\
&= \mathbf{L}\ \mathit{target}\ \mathit{input}
\end{aligned}
$$

Verification that program *compiler* translates source programs into equivalent target programs is also straightforward:

$$
\begin{aligned}
\mathit{target} &= \mathbf{L}\ \mathit{mix}\ (\mathit{int},\ \mathit{source}) \\
&= \mathbf{L}\ (\mathbf{L}\ \mathit{mix}\ (\mathit{mix},\ \mathit{int}))\ \mathit{source} \\
&= \mathbf{L}\ \mathit{compiler}\ \mathit{source}
\end{aligned}
$$

Finally, we can see that *cogen* transforms interpreters into compilers by the following:

$$
\begin{aligned}
\mathit{compiler} &= \mathbf{L}\ \mathit{mix}\ (\mathit{mix},\ \mathit{int}) \\
&= \mathbf{L}\ (\mathbf{L}\ \mathit{mix}\ (\mathit{mix},\ \mathit{mix}))\ \mathit{int} \\
&= \mathbf{L}\ \mathit{cogen}\ \mathit{int}
\end{aligned}
$$

See [12] for a more detailed discussion.

2.4 Efficiency in Practice

A variety of partial evaluators satisfying all the above equations have been constructed ([12, 4, 13] contain more detailed discussions). Compilation, compiler generation and compiler generator generation can each be done in two ways, e.g.

$$
\mathit{target} = \mathbf{L}\ \mathit{mix}\ (\mathit{int},\ \mathit{source}) = \mathbf{L}\ \mathit{compiler}\ \mathit{source}
$$

and similarly for generation of *compiler* and *cogen*. Although the exact timings vary according to the partial evaluator and the implementation language **L**, in our experience the second way is often about 10 times faster than the first (for all three cases). A less machine dependent and more intrinsic efficiency measure will be seen in section 4.

3 A Concrete Programming Language

We now describe the Lisp-like language used in our experiments with the s-m-n and recursion theorems, discuss program efficiency and outline a universal program.

3.1 The Language Mixwell

In Mixwell both programs and data are S-expressions, i.e. elements of D. A Mixwell program is the representation of a system of recursive equations as an element of D, with syntax

```
((f1 (x1 x2 ... xn) = expression1)
 (f2 (y1 y2 ... yp) = expression2)
  ...
((fm (z1 z2 ... zq) = expressionm))
```

Here `expression1`, `expression2`, etc. are constructed from variables (e.g. `y2`) and constants of form `(quote d)` where d is an element of D (for brevity, `(quote d)` is written as `'d` in examples). The operations allowed in expressions include application of base functions (arithmetic and comparison operations, constructors and destructors for D, etc.), conditionals `(if-then-else)` and calls to the user-defined functions `f1`, `f2`,.... As usual in Lisp a function call has form `(function-name argument...argument)`. For an example, consider a program to compute the function x^n (different from the one given before).

```
((p(n,x) =  (if (zero? n)  then '1 else
            (if (even? n)  then (square  (p (divide n '2) x)) else
                                (times x (p (subtract1 n) x)))))
```

Program semantics is as usual for statically scoped Lisp. Details are beyond the scope of this paper, but briefly: the program's meaning is the meaning of user-defined function `f1`; and call by value is used for all function calls (i.e. arguments of a call to `fi` are evaluated before the expression that defines `fi` is evaluated). It is easy to show that Mixwell is an acceptable programming system [1].

3.2 Outline of a Self-interpreter

The universal program *up* of Definition 2.1 is easy to program in Mixwell. Figure 1 contains a sketch of the natural self-interpreter:

```
((up (program data) =
    (eval (4th (1st program))     ; right side of the first equation
          (2nd (first program))   ; its list of formal parameter names
          (list data)             ; list of formal parameter values
          program)                ; the program (used in function calls)
 )
 (eval (exp parnames values prog) =
    (if (quote? exp)
                then (2nd exp) else
    (if (variable? exp)
                then (lookupvalue exp parnames values) else
    (if (subtract1? exp)
                then (subtract1 (eval (2nd exp) parnames values program))
                else
    (if ...     then ...
                else (quote SYNTAX-ERROR) )))) )
 (lookupvalue (exp parnames values prog) =  ...) )
```

Figure 1: A universal program for Mixwell

Explanation Here we have used base function `1st` to select element d_1 from the list $(d_1 d_2 \ldots d_n)$, `2nd` to select element d_2, etc. (all can be expressed via Lisp's two primitives `car, cdr`). The interpreter's central function is `eval`, which is given as arguments:

an expression to evaluate; the names of the parameters to the function currently being evaluated; the values of those parameters; and the text of the program being executed.

`eval` works by determining which of the allowable forms the expression has, and then taking the appropriate evaluation actions. If it has form `(quote d)`, then its value is `d`—the second component of `exp`, computed by `(2nd exp)`. If a variable, `eval` calls function `lookupvalue` to locate the variable in list `parnames`, and to extract the corresponding value from list `values`. If of form `(subtract1 e)`, then `eval` calls itself recursively to get the value of `e`, and the subtracts 1 from the result; and analogous computations are done for all the other cases.

The main function `up` simply calls `eval` with the appropriate arguments—the right side of the first equation, the name of its single argument, that argument's value `data`, and the whole program `prog`.

4 Techniques and Efficiency of Partial Evaluation

From the Futamura projections it is not at all clear how efficient we can expect *mix*-produced specialized programs to be, and it is also unclear how to measure the quality of specialized programs. For practical purposes, even a small speedup can be profitable if a program is to be run often. Further, it can be faster to compute even a single value $f(d_1, d_2)$ in two steps: first, specialize f's algorithm to d_1; and then run the specialized program on d_2. (A familiar example: compiling a Lisp program and then running the result is often faster than running Lisp interpretively.)

Since increased efficiency is the prime motivation for using partial evaluation on the computer, we now discuss techniques for gaining efficiency, and motivate the development of a more abstract and general understanding of efficiency.

4.1 Goals of Partial Evaluation

Let p be a program with two input parameters. Then $\mathbf{L}\ p\ (d_1, d_2)$ denotes the value gotten by running p on (d_1, d_2). If only d_1 is available, evaluation of $\mathbf{L}\ p\ (d_1, \_)$ does not make sense, as the result is likely to depend on d_2. However, d_1 might be used to perform *some* of the computations in p, yielding as result a transformed, optimized version of p.

The goal of a practical partial evaluator is thus: to analyze its subject program in order to find out which of its calculations may be performed on basis of the incomplete input data d_1; to perform them; and to construct a specialized program containing only those computations essentially dependent on d_2.

4.2 Some Techniques for Partial Evaluation

Successful self-applicable partial evaluators include [12], [5], [4] and [13]. The techniques used there include: applying base functions to known data; unfolding function calls; and creating versions of program functions which are specialized to data values computable from the known input d_1.

To illustrate these techniques, consider the well-known example of Ackermann's function (using the same informal syntax as in the introductory section):

```
a(m,n) =  if m=0 then n+1 else
```

```
if n=0 then a(m-1,1)
       else a(m-1,a(m,n-1))
```

Computing `a(2,n)` involves recursive evaluations of `a(m,n')` for `m` = 0, 1 and 2, and various values of `n'`. The partial evaluator can evaluate `m=0` and `m-1` for the needed values of `m`, and function calls of form `a(m-1,...)` can be unfolded (i.e. replaced by the right side of the recursive equation above, after the appropriate substitutions).

We can now specialize function `a` to the values of `m`, yielding the residual program:

```
a2(n) =  if n=0 then 3 else a1(a2(n-1))
a1(n) =  if n=0 then 2 else a1(n-1)+1
```

This program performs less than half as many arithmetic operations as the original since all tests on and computations involving `m` have been removed. The example is admittedly pointless for practical use due to the enormous growth rate of Ackermann's function, but it illustrates some simple and important optimization techniques.

4.3 Program Running Times

A reasonable approximation to program running times on the computer can be obtained by counting 1 for each of the following: constant reference, variable reference, test in a conditional or case, function parameter, and base or user-defined function call. Thus the exponentiation example has time estimate:

$$t(n,x) = \begin{cases} 4 & \text{if } n = 0, \text{ else} \\ 14 + t(n/2, x) & \text{if } n \text{ even, else} \\ 14 + t(n-1, x) & \end{cases}$$

which can be shown to be of order $\log n$.

4.4 The Computational Overhead Caused by Interpretation

On the computer, interpreted programs are often observed to run slower than compiled ones. We now explain why, using the previously described self-interpreter as an example.

Let $t_p(d)$ denote the time required to calculate **L** p d. As seen in section 3.2 the basic cycle of the self-interpreter *up* is first syntax analysis: a series of tests to determine the main operator of the current expression to be evaluated; then evaluation of necessary subexpressions by recursive calls to `eval`; and finally, actions to perform the main operator, e.g. to subtract 1 or to look up the current value of a variable. It is straightforward to see that the running time of *up* on list $(p\,d)$ satisfies

$$at_p(d) \leq t_{up}((p\,d))$$

for all d, where a is a constant. In our experiments a is often around 10 for small source programs. (In this context "constant" means: a is independent of d, although it may depend on p.)

4.5 Efficiency of Partial Evaluation, Interpretation and Compilation

Program speedup is not a universal phenomenon, and one can only expect $s_1^1(p, d_1)$ to be significantly faster when p's control and subcomputations are largely determined by d_1. For example, specializing the exponentiation algorithm to its second argument would not decrease its running time. On the other hand for an interpreter, d_1 is the program being interpreted, and its structure strongly directs the computation.

The speedup of the previous section is typical in our experience: an interpreted program runs slower than one which is compiled (or executed directly, which amounts to being interpreted by hardware); and the difference is a linear factor, large enough to be worth reducing for practical reasons, and depending on the size of the program being interpreted. Further, clever use of data structures (hash tables, binary trees, etc) can make a grow slowly as a function of p's size.

A Goal for Compilation by Partial Evaluation

For practical purposes the trivial partial evaluation given by the traditional s-m-n construction (e.g. as illustrated in the introduction) is uninteresting; in effect it would yield a target program of the form "apply the interpreter to the source program text and its input data". Ideally, *mix* should remove *all computational overhead* caused by interpretation.

On the other hand, how can we meaningfully assert that a partial evaluator is "good enough"? Perhaps surprisingly, a machine-independent answer can be given. This answer involves the mix equation and the self-interpreter *up* seen earlier. For any program p

$$\begin{aligned} \mathbf{L}\ p\ d &= \mathbf{L}\ up\ (p,\ d) \\ &= \mathbf{L}\ (\mathbf{L}\ mix\ (up,\ p))\ d \end{aligned}$$

so $p' = \mathbf{L}\ mix\ (up,\ p)$ is an **L**-program equivalent to p. This suggests a natural goal: that p' be at least as efficient as p. Achieving this goal implies that *all computational overhead* caused by *up*'s interpretation has been removed by *mix*.

Definition 4.1 *Mix is* optimal *on up provided* $t_{p'}(d) \leq a + t_p(d)$ *for some constant* a *and all* $p, d \in D$, *where* $p' = \mathbf{L}\ mix\ (up,\ p)$.

We have satisfied this criterion on the computer for several partial evaluators for various languages, using self-interpreters such as the one sketched in section 3.2. For some of these p' is identical to p (up to variable renaming).

Discussion *Mix* is a general specializer, applying uniform methods to all its program inputs. In practice the input program must be written in a straightforward manner, so *mix* can succeed in analyzing the way the control and computations of its program input p depend on the available data d_1. A special case: the universal program *up* must be "cleanly" written for *mix* to be optimal.

It can occur that $p' = \mathbf{L}\ mix\ (up,\ p)$ is *more* efficient than p, e.g. if p is written with redundant computations. This is the exception rather than the rule, though—it is

unreasonable to expect speedup on all programs. **Question:** how can this be precisely formulated and proven?.

Unfortunately, one can "cheat" the optimality criterion above. For example, consider a *mix* algorithm that begins by testing its program input to see whether it is the specific universal program *up* from section 2.3. If so, then this version of *mix* just outputs its second argument without change; if not, it realizes a trivial partial evaluation. Such a partial evaluator would satisfy the letter of the above definition of optimality but not its spirit. (Disclaimer: our optimal *mix* es don't cheat.)

Open problem: Find a better definition of optimality that rules out "cheating".

4.6 Desirable: a Complexity Theory for Partial Evaluation

Partial evaluation is chiefly concerned with *linear* time speedups—and the now well-developed theory of computational complexity [7] traditionally ignores linear factors (perhaps due to the abundance of unsolved open problems even bout larger increases in computing time). In any case, the open problems of complexity theory concern lower complexity bounds on specific problems, and these are especially difficult since a lower bound is less than or equal to the complexity of the best of *all possible* algorithms that solve the problem, regardless of the techniques employed.

On the other hand the s_1^1 function used for partial evaluation is a *uniform* program specializer that works on any program at all, and so is not problem-specific. Further, significant efficiency increases (and even optimality as defined above) can be accomplished with a rather limited set of program transformations. This gives some hope for a complexity theory of partial evaluation, since uniformity implies that the specializer is not expected to be sensitive to the perhaps mathematically sophisticated computation strategies used by the program being specialized.

We now give a simple question that we hope could be formulated precisely and solved, given a suitable complexity theory for partial evaluation. As before, let $t_p(d)$ be the time to compute $\mathbf{L}\ p\ d$, and let $p_{d_1} = \mathbf{L}\ mix\ p\, d_1$ be the result of specializing program p to known input d_1.

For a trivial partial evaluator as in the standard s-1-1 construction, one would expect

$$t_{p_{d_1}}(d_2) = t_p((d_1\, d_2)) + b$$

for constant $b \geq 0$, where b is the "set-up time" required to initialize p's first argument to d_1.

In our experiments (chiefly on interpreters, parsers and string matchers), p_{d_1} often runs substantially faster than p by factors ranging from 3 to 50, and speedups of over 200 have been reported in the literature.

Definition 4.2 *We say that mix accomplishes* linear speedup *on p if for all $d_1 \in D$ there is a function $f(n) = an$ with $a > 1$ such that for all $d_2 \in D$*

$$f(t_{p_{d_1}}(d_2)) \leq t_p((d_1\, d_2))$$

To clarify this definition, consider two examples. First, let p be an interpreter. We saw in the previous section that partial evaluation can sometimes eliminate all interpretation

overhead, e.g. when applied to a self-interpreter. *A finer analysis*: the interpretation overhead involves two factors. One is independent of p and represents the time for the syntax analysis part of the "interpretation loop" together with the time to perform elementary operations. The other depends on p and is a (usually) slowly growing function representing the time required to fetch and/or store variables' values. Thus the exact speedup factor a will depend on the value of the program being interpreted, but the speedup is still linear by the definition above.

For a second example, let p be a string matching program that searches for an occurrence of pattern string d_1 in subject string d_2. By a naive algorithm this takes time $t_p((d_1\, d_2)) = O(mn)$ where m, n are the lengths of d_1 and d_2, respectively. The well-known Knuth-Morris-Pratt string matching algorithm yields an algorithm p_{d_1} that runs in time $O(n)$. In this case the matching time $t_{p_{d_1}}(d_2)$ is independent of d_1. A linear speedup with factor a as large as desired is thus obtainable by choosing d_1 to make m large enough.

Consel and Danvy have shown in [6] that partial evaluation, given a simple matching program p and pattern d_1, yields an algorithm that runs in time $O(n)$. Thus the speed of a sophisticated string matcher can be obtained by partially evaluating a fairly naive program, avoiding the need for the insight shown by Knuth, Morris and Pratt.

Open question If *mix* uses only the techniques given earlier in this section and preserves termination properties, do there exist programs p on which *mix* accomplishes superlinear speedups? *Discussion*: in our experience, speedups of interpreters and other programs are always linear in the above sense. Clearly, superlinear speedups can be obtained by changing p's computational algorithm, or by discarding unnecessary computations. On the other hand, algorithm changes would seem to involve nonuniform transformations; and discarding computations can change a nonterminating program into a terminating one. It thus remains unclear whether uniform techniques can make superlinear optimizations.

5 Efficient Realization of the Recursion Theorem

In programming terms, Kleene's "second recursion theorem" [14] says that programs may without loss of computability be allowed to reference their own texts. This powerful theorem has been used frequently in recursive function theory and in "machine-independent" computational complexity theory ([19, 15, 3]). Although its content is constructive, the theorem seems most often used negatively—as a tool for proving various hierarchies nontrivial by showing the existence of "pathological" functions and sets having great complexity or unexpected properties.

This section deals with a practical and, as it turns out, reasonably efficient implementation of the recursion theorem. We give a brief review of the fundamentals, then discuss our implementation of the theorem and report on two experiments with it; more may be seen in [1].

Theorem 5.1 (The second recursion theorem) *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 for any program, $p \in D$

$$\varphi_p(y, x) = \varphi_{s_1^1(p,y)}(x)$$

It is evidently possible to construct a program $q \in D$ such that

$$\varphi_q(y, x) = \varphi_p(s_1^1(y, y), x)$$

Let e be the program $s_1^1(q, q)$. Then we have

$$\varphi_p(e, x) = \varphi_p(s_1^1(q, q), x) = \varphi_q(q, x) = \varphi_{s_1^1(q,q)}(x) = \varphi_e(x)$$

□

5.1 Some Applications

Following are some typical results which are easily proven using the recursion theorem, and which would seem rather difficult without it. All are language independent since they hold for any acceptable programming system.

Self-reproduction Let $r \in D$ be a program with $\varphi_r(p, x) = p$. According to theorem 5.1, there is a program e such that for all x

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

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

Eliminating recursion Consider the computable function

$$f(p, x) = [\texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * \varphi_{up}(p, x - 1)]$$

By theorem 5.1, f has a fixed-point e with the property

$$\begin{gathered}\varphi_e(x) = f(e, x) = \\ [\texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * \varphi_{up}(e, x - 1)] = \\ [\texttt{if}\ x = 0\ \texttt{then}\ 1\ \texttt{else}\ x * \varphi_e(x - 1)]\end{gathered}$$

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 ways the theorem has been used constructively to solve computationally interesting problems.

Unsolvability The second recursion theorem can be used to give a very elegant proof (see [15] or [19]) of

Theorem 5.2 (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.*

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*, $\psi_p(d)$ giving the resources used when running program p on input d, satisfying natural requirements stated in [3]. (Running time is one example, and memory usage is another.)

Theorem 5.3 (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*

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

for almost all n.

A program computing f can be found constructively from one for r, but going from program p for f to the "faster" program q is a non-constructive operation. The theorem is therefore of limited interest for practitioners.

5.2 A Classical Implemention of the recursion theorem

Following the recipe implied by the following classical proof, we wrote a Mixwell program that when given another program as input, computes its Kleene fixed-point.

The method works on the computer but has some drawbacks. First, the fixed-point programs are rather large, since they contain both the whole input program text plus code (for the s-m-n function etc.) that in effect allows the fixed-point program to reference its own text. Second, applications often involve interpretation. i.e. applying the universal function. Thus *up* must be present in the fixed-point program, so a direct implementation quickly becomes inefficient since every nested call to *up* results in an extra level of interpretation.

For example running a fixed-point program to compute $n!$ as in the example given above results in n levels of interpretation. The run time is exponential in n, since each level slows down execution by a large constant factor (around 10 in the example above). Lesson: a too direct implementation is not practical.

5.3 The Reflect language

To overcome these efficiency problems we devised a new language 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 $\varphi_p(i)$.

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

Implementation Each `(univ ...)` expression gives rise to a call of the same interpreter that is used to implement Reflect, and thus adds no extra levels of interpretation. The special expression `*` is quite naturally available, e.g. in section 3.2 it is the variable `prog`.

Reflect indeed has the power to express fixed-points, by the construction shown below. Its correctness proof (given in [1]) relies on a near-denotational semantics of the language, which is an almost trivial extension of the truly denotational semantics of Mixwell. It is an easy exercise to prove

Lemma 5.4 Reflect *is an acceptable programming system.*

Theorem 5.5 *Given a* Reflect *program p with the following structure:*

```
((p1 (f x) = expression1)
 (p2 (...) = ...  )
  ... )
```

The following Reflect *program e is a Kleene fixed-point for p:*

```
((fix (x) = (univ '((p1 (f x) = expression1)
                    (p2 (...) = ...  )
                     ... )
                   (list * x) )))
```

This avoids the nesting of interpretation levels (and the corresponding multiplication of running times) that occurred when applying the traditional Kleene construction. The reason is that by the way Reflect is implemented, applications of the universal function within p cause only a limited amount of extra overhead. Further, the time to run the program above on x is just a constant plus the time to run p on e and x.

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. Consequently, self-reproducing programs are very easily expressed in Reflect:

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

Running the program produces exactly the text shown above.

Example 2 The effect of recursion can also be obtained nonrecursively in Reflect:

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

On the computer, the time to compute $n!$ grows linearly in n, so a major improvement has occurred over the direct implementation. The same holds if $n!$ is computed as in Theorem 5.5.

5.4 Remarks and open questions

We designed a language Reflect that allows fixed-point programs to be expressed naturally and executed with reasonable efficiency, giving a framework for experiments with constructive applications of the second recursion theorem.

Is it cheating to define Reflect as a language with built-in self-reference? In our opinion the answer is no, for two reasons. First, Mixwell programs run just as fast under the Reflect interpreter as before; one would expect a slowdown if we had done something computationally unrealistic. (The reason is that the Reflect interpreter is simply a small extension of that for Mixwell.) Second, we can compute $n!$ and other reflexively defined functions *much* more efficiently in Reflect than by the classical construction from Kleene's theorem (either directly, as in the example above, or by the general construction of Theorem 5.5).

The most pressing open problem in this direction is to widen the spectrum of computationally interesting applications. Further, it seems likely that the second recursion theorem implies the existence of an efficiently implementable extensible programming language, but this possibility has not yet been realized in practice.

References

[1] Torben Amtoft, Thomas Nikolajsen, Jesper Larsson Träff, Neil D. Jones: Experiments with Implementations of two Theoretical Constructions. Logic at Botik, Lecture Notes in Computer Science,Springer-Verlag, vol. 363, pp. 119-133, 1989.

[2] A. Appel: Semantics-Directed Code Generation. 12th ACM Symposium on Principles of Programming Languages, pp. 315-324,1985.

[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] Anders Bondorf: Automatic Autoprojection of Higher Order Recursive Equations. CAAP/ESOP1990 Proceedings, LNCS 432, Springer-Verlag, 1990.

[5] A. Bondorf, O. Danvy: Automatic Autoprojection of Recursive Equations with Global Variables and Abstract Data Types. DIKU report 90-4, University of Copenhagen,1990.

[6] C. Consel, O. Danvy: Partial Evaluation of Pattern Matching in Strings. Information Processing Letters, Vol. 30, No 2, pp 79-86, January 1989.

[7] S. Cook: An Overview of Computational Complexity. Communications of the ACM 26, pp. 401-408,1983.

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

[9] Y. Futamura: Partial Evaluation of Computation Process—an Approach to a Compiler-compiler. Systems, Computers, Controls, 2(5), pp. 45-50, 1971.

[10] G. Goguen, J. W. Thatcher, E. G. Wagner, J. B. Wright: Initial Algebra Semantics and Continuous Algebras. Journal of the ACM vol. 24, pp. 68-95, 1977.

[11] C. K. Gomard: Higher Order Partial Evaluation: H.O.P.E. for the Lambda Calculus. M. S. thesis, DIKU, University of Copenhagen, 1989.

[12] 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.

[13] N. D. Jones, C. K. Gomard, A. Bondorf, O. Danvy, T. Mogensen: A Self-applicable Partial Evaluator for the Lambda Calculus. IEEE Computer Society 1990 International Conference on Computer Languages, pp. 49-58,1990.

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

[15] A. I. Mal'cev: Algorithms and recursive functions. McGraw-Hill, 1967.

[16] F. L. Morris: Advice on Structuring Compilers and proving them correct. 1st ACM Symposium on Principles of Programming Languages, pp. 144-152, 1973.

[17] P. Mosses: SIS—Semantics Implementation System, Reference Manual and User Guide. DAIMI Report MD-30, University of Aarhus, Denmark, 1979.

[18] L. Paulson: A Semantics-Directed Compiler Generator. 9th ACM Symposium on Principles of Programming Languages, pp. 224-233, 1982. Wolters-Noordhoff Publishing, 1970.

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

[20] D. Schmidt: Denotational Semantics: a Methodology for Language Development. Allyn and Bacon, 1986

[21] V. F. Turchin: The Concept of a Supercompiler. ACM Transactions on Programming Languages and Systems, 8(3), pp. 292-325, 1986.

[22] P. Weis: Le Systeme SAM: Metacompilation tres efficace a l'aide d'operateurs semantiques. Universite de Paris 7, 1987.