Efficient Algebraic Operations on Programs

Neil D. Jones

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

Abstract

A *symbolic version* of an operation on values is a corresponding operation on program texts. For example, symbolic composition of two programs `p`, `q` yields a program whose meaning is the (mathematical) composition of the meanings of `p` and `q`. Another example is symbolic specialization of a function to a known first argument value. This operation, given the first argument, transforms a two-input program into an equivalent one-input program. Computability of both of these symbolic operations has long been established in recursive function theory [12,16]; the latter is known as Kleene's "s-m-n" theorem, also known as *partial evaluation.*

In addition to computability we are concerned with *efficient* symbolic operations, in particular applications of the two just mentioned to compiling and compiler generation. Several examples of symbolic composition are given, culminating in nontrivial applications to compiler generation [14], [18]. Partial evaluation has recently become the subject of considerable interest [1]. Reasons include simplicity, efficiency and the surprising fact that self-application can be used in practice to generate compilers, and a compiler generator as well.

This paper makes three contributions: First, it introduces a new notation to describe the types of symbolic operations, one that makes an explicit distinction between the types of program texts and the values they denote. This leads to natural definitions of what it means for an interpreter or compiler to be type correct—a tricky problem in a multilanguage context. Second, it uses the notation to give a clear overview of several earlier applications of symbolic computation. For example, it is seen that the new type notation can satisfactorily explain the types involved when generating a compiler by self-applying a partial evaluator. Finally, a number of problems for further research are stated along the way. The paper ends by suggesting Cartesian categorical combinators as a unifying framework in which to study symbolic operations.

1 Introduction

When solving a mathematical problem using a computer one usually thinks first about the subject area: operations on values, functions, matrices, logical formulas, etc.; and only later about how they are to be realized algorithmically. Often, however, a straightforward implementation of a problem statement is too inefficient for practical use, leading to the need for so-called "optimization" to find more efficient computer solutions. Optimization can take many forms. Some of the most powerful speedups result from reformulating the problem—finding a mathematically equivalent restatement that is computationally better. Unfortunately, such methods can be quite subtle, requiring a deep problem understanding that takes years to develop.

An alternative to redoing the mathematics is to let the computer itself find a more efficient implementation of a "naive" algorithm. This leads to the goal of using the computer to transform one program into a new and equivalent but more efficient program.

1.1 High-level Operations in Programming Languages

Programming languages of higher and higher abstraction levels have evolved since the early years of computing (when programming languages were just symbolic codes reflecting the computer's architecture!). Due to higher level basic operations, modern functional languages allow a mathematical style of thinking while programming, for example using function composition, partial function application, set comprehension and pattern matching. This is possible since these mathematical operations are known to give computable results when applied to computable arguments.

On digital computers mathematical operations are specified by textual objects, e.g. programs and their parts, for example expressions. It is well known that many mathematical operations can be faithfully realized by operations on symbolic expressions. A classic example is algebraic symbol manipulation, which abstractly but correctly describes concrete operations on numbers, matrices, etc. The term "symbolic computation" often refers to such manipulations when realized on the computer, but can equally well be interpreted more broadly: to describe the entire theme of this article.

1.2 Operations on Functions and Programs

Two useful operations on (mathematical) functions are:

- *Composition* of f with g, written as $f; g$ or $g \circ f$.
- *Function specialization* of $f(x, y)$, obtaining a one-argument function by "freezing" x to a fixed value, e.g. $x = a$.

Corresponding symbolic operations on programs (assuming for concreteness that they are written in the λ-calculus):

- *Symbolic composition*: suppose f, g are computable by expressions e_f, e_g. Then their composition is computable by expression $\lambda x.e_g(e_f(x))$
- *Partial evaluation*: the specialization of function f to $x = a$ can be realized symbolically as the program $\lambda y.e_f(a, y)$. A well-known version of the same result, in the context of recursive function theory, is Kleene's s-m-n theorem [12,16].

1.3 Efficient Operations on Programs

The symbolic operations above have a common characteristic: while computable, they do not lead to particularly efficient programs. For example the composition program is no faster than just running the two programs from which it is constructed, one after the other. The main theme of this article is the *efficient* implementation of program operations that realize mathematical operations, in particular function composition and specialization. We begin with an example of each.

Composition

Consider the composition *sum* $\circ$ *squares* $\circ$ *oneto*, where $oneto(n) = [n, n-1, \ldots, 2, 1]$ yields a list of the first n natural numbers, $squares[a_1, a_2, \ldots a_n] = [a_1^2, a_2^2, \ldots, a_n^2]$ squares each element in a list, and $sum[a_1, a_2, \ldots a_n] = a_1 + a_2 + \ldots + a_n$ sums a list's elements. A straightforward program to compute *sum* $\circ$ *squares* $\circ$ $oneto(n)$ is:

```
f(n)       =  sum(squares(oneto(n)))
squares(l) =  if l = [] then [] else cons(head(l)**2, squares(tail(l)))
sum(l)     =  if l = [] then [] else head(l) + sum(tail(l))
oneto(n)   =  if n = 0  then [] else cons(n, oneto(n-1))
```

A significantly more efficient and "listless" program [21] for *sum* $\circ$ *squares* $\circ$ *oneto* is:

```
g(n) = if n = 0 then 0 else n**2 + g(n-1)
```

Function Specialization

Consider 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)
```

A straightforward specialization of the program above to $\texttt{n} = 5$ is 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)
```

This corresponds to the trivial λ-calculus construction given above, or the classical proof of Kleene's s-m-n theorem [12]. A better program for $\texttt{n} = 5$ can be got by unfolding applications of the function `p` and doing all computations involving `n` and reducing `x*1` to `x`, yielding the residual program:

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

1.4 Program Running Times

How can we measure the efficiency increase of the examples just given? Assuming call by value and traditional implementation techniques, a reasonable approximation to program running times on the computer can be obtained as follows. Count 1 for each constant reference, variable reference, test in a conditional or case, function parameter, and base or user-defined function call.

Thus the exponentiation program `p` above has time estimate:

$$\begin{array}{ll} t_{\texttt{p}}(n,x) = 4 & \text{if } n = 0\text{, else} \\ \qquad 15 + t_{\texttt{p}}(n/2,x) & \text{if n even, else} \\ \qquad 15 + t_{\texttt{p}}(n-1,x) & \end{array}$$

which can be shown to be of order $\log n$. For this specific program and this counting scheme, $t_{\texttt{p}}(5,x) = 65$. On the other hand the specialized program has running time 7. Similarly, for input n the result of the symbolic composition example takes time $4 + 11 \cdot n$ whereas the "naive" version takes time $15 + 32 \cdot n$.

1.5 Goals of this Article

Our main goals are:

1. To review some nontrivial symbolic operations on programs that have been successfully realized on the computer.
2. To put earlier results into a broader perspective.
3. To list some challenging problems for further study in the area.
4. To propose a more general study of symbolic operations on programs.

Our context of discourse is the following (worth making explicit since it differs from those of many other works about programming languages):

- We consider programs as data objects, and will not discuss their internal structure. (Internal structures and their semantics would be unavoidable if we were discussing *how* to do symbolic composition, etc., but that would be the subject of another paper.)
- Programs' operational behavior will be emphasized, i.e. observable results when applied to first order inputs and outputs.

- We seek a framework for *automatic* and *general* methods, as opposed to one-at-a-time or hand construction of specific program transformations.
- Compiling and compiler generation are of particular interest. This implies a multilanguage context.
- We distinguish between the types of *values* and those of *programs*. For example the number 3 has type *integer*, but Lisp program `(+ 1 2)` has another type ($\underline{integer}_{\mathbf{Lisp}}$, to be explained later).
- Programs will be considered not as untyped but as *polytyped*; for example, a correct compiler translates a source program denoting any expressible type into a target program denoting the same type.

Acknowledgements

This article has been inspired by discussions with (among others) the Topps group at DIKU, John Hughes, John Launchbury, and Alan Mycroft.

2 Definitions and Terminology

We seek a framework to discuss among other things:

- symbolic composition and other operations on programs,
- correctness of compilers, interpreters, and partial evaluators, and
- the types of compilers, interpreters, and partial evaluators.

Compilers and interpreters involve more than one language, so we must first find suitable definitions of programming languages, program meanings, and the values programs manipulate.

Languages will be denoted by Roman letters, e.g. **L** (a default “meta-” or “implementation” language), **S** (a “source language” to be interpreted or compiled) and **T** (a “target language” that is the output of a compiler or partial evaluator). In this paper a program will be assumed to take zero, one, or more first order inputs. An observable result may be a single output, or failure to terminate. We do not consider nondeterminism, communication or other advanced features.

First Order Data

Programs will be treated as data objects, so it is natural to draw both program texts and their input/output data from a common *first order data* domain D. Its details are unimportant for this paper, the only essential feature being that any program can be considered as an element of D. For example one could follow the well-established Lisp tradition using the set of “S-expressions”[1] for D[2].

Higher Order Values

We will also need higher order values; for example a program's meaning is often a function from some number of first order inputs to a first order output. It is convenient to assume that the number of a program's inputs is not fixed in advance. A motivating example: an interpreter takes as input both a program to be interpreted, and *its* input as well. The ability to interpret programs with different numbers of inputs requires that the interpreter itself can accept input sequences of varying lengths[3].

We use a universal value domain V defined as follows. Appropriate mathematical interpretations are described in [17] and other works on denotational semantics. As is customary we omit summand projections and injections.

[1]Defined as the least set such that $D = Atom \cup D \times D$, where an “atom” is any sequence of one or more letters, digits or characters excluding parentheses, comma and blank.

[2]In fact any countably infinite set would do, e.g. the natural numbers as in recursive function theory.

[3]An alternative is to assume that every function has exactly one input, possibly structured. We have tried this but found it to be notationally heavier when defining interpreters, partial evaluators, etc.

$$V = D_\perp + (D \to V)$$

There are three sorts of values: a first order value in D, the undefined value ($\perp$ indicates nontermination), or a function. If a value is a function, its application to an argument yields a value, again of any of these three sorts[4].

Multiple function arguments are handled by "currying". For example if $f \in V$ is a program meaning with inputs $\mathtt{d}_1, \ldots, \mathtt{d}_n \in D$ then $f\,\mathtt{d}_1\,\mathtt{d}_2 \ldots \mathtt{d}_n$ (associated from the left) is the result of applying f to $\mathtt{d}_1$ yielding a function which is then applied to $\mathtt{d}_2$, etc.

2.1 Programming Languages

Definition 2.1 *A* programming language *is a function* $[\![\_]\!]_{\mathbf{L}} : D \to V$ *that associates with each* $\mathtt{p} \in D$ *a value* $[\![\mathtt{p}]\!]_{\mathbf{L}} \in V$*. The subscript* $\mathbf{L}$ *will be omitted when clear from context.*

Intuitively, a language $\mathbf{L}$ is identified with its "semantic function" $[\![\_]\!]_{\mathbf{L}}$ on whole programs. *All* elements $\mathtt{p}$ of D are regarded as programs; ill-formed programs can for example be mapped to an everywhere undefined function.

Notational convention. We associate function application from the left and use as few parentheses as possible, for example writing $([\![\mathtt{p}]\!]_{\mathbf{L}}(\mathtt{d}_1))(\mathtt{d}_2)$ as $[\![\mathtt{p}]\!]_{\mathbf{L}}\ \mathtt{d}_1\ \mathtt{d}_2$, or just $[\![\mathtt{p}]\!]\ \mathtt{d}_1\ \mathtt{d}_2$ if $\mathbf{L}$ is understood from context.

Following is enough syntax to describe the result of executing programs on first order data. (It also includes some operationally meaningless syntax, e.g. $\mathtt{3(}[\![\mathtt{p}]\!]_{\mathbf{L}}\mathtt{)}$.)

Definition 2.2 Abstract syntax for program runs:

exp	::=	*first-order-data-value* | $exp_1\ exp_2$ | $[\![exp]\!]_{\mathbf{X}}$
first-order-data-value	::=	*... (unspecified)*

Examples for $\mathbf{L}$ = Lisp:

$$[\![\texttt{(quote ALPHA)}]\!]_{\mathbf{L}} = \text{ALPHA}$$
$$[\![\texttt{(lambda (x) (+ x x))}]\!]_{\mathbf{L}}\ 3 = 6$$

2.2 Data and Program Types

We now extend the usual concept of type.

Definition 2.3 *The* Abstract syntax *of a type t:*

$$t\text{: } type \ ::= \ \underline{type}_{\mathbf{X}} \mid firstorder \mid type \times type \mid type \to type$$

Type *firstorder* describes values in D, i.e. S-expressions, and function types and products are as usual. For each language $\mathbf{X}$ and type t we have a type constructor, written $\underline{t}_{\mathbf{X}}$ and meaning the type of all $\mathbf{X}$-programs which denote values of type t. For example, atom `ALPHA` has type *firstorder*, and Lisp program `(quote ALPHA)` has type $\underline{firstorder}_{\mathbf{Lisp}}$.

Type Deduction Rules Figure 1 contains some type deduction rules sufficient to describe program runs, i.e. evaluations of closed expressions.

[4]Note that this definition allows a function to have varying numbers of arguments.

$$\frac{\mathrm{exp}_1 : \mathrm{t}_2 \rightarrow \mathrm{t}_1, \quad \mathrm{exp}_2 : \mathrm{t}_2}{\mathrm{exp}_1\mathrm{exp}_2 : \mathrm{t}_1} \qquad \frac{\mathrm{exp} : \underline{t}_{\mathbf{X}}}{[\![\mathit{exp}]\!]_{\mathbf{X}} : \mathrm{t}}$$

$$\frac{}{\mathit{firstordervalue} : \mathit{firstorder}} \qquad \frac{\mathrm{exp} : \underline{t}_{\mathbf{X}}}{\mathit{exp} : \mathit{firstorder}}$$

Figure 1: Some Type Inference Rules for Closed Expressions

Remark An object `p` of type $\underline{t}_{\mathbf{X}}$ is a program text and thus *in itself* a value in D, i.e. $\underline{t}_{\mathbf{X}}$ denotes a subset of D. On the other hand, `p`'s denotation $[\![\mathtt{p}]\!]_{\mathbf{X}}$ may be any value in V, for example a higher order function.

Semantics of Types

Definition 2.4 *The meaning of type expression t is $[\![t]\!]$ defined as follows:*

$$\begin{array}{lcl}
[\![\mathit{firstorder}]\!] & = & D \\
[\![t_1 \rightarrow t_2]\!] & = & [[\![t_1]\!] \rightarrow [\![t_2]\!]] \\
[\![t_1 \times t_2]\!] & = & \{(t_1, t_2) \mid t_1 \in [\![t_1]\!],\ t_2 \in [\![t_2]\!]\} \\
[\![\underline{t}_{\mathbf{X}}]\!] & = & \{\mathtt{p} \in D \mid [\![\mathtt{p}]\!]_{\mathbf{X}} \in [\![t]\!]\}
\end{array}$$

Polymorphism

We will also allow *polymorphic* type expressions to be written containing *type variables* $\alpha, \beta, \gamma, \ldots$. Such a polymorphic type will always be understood relative to a substitution mapping type variables to type expressions without variables. The result of applying the substitution is called a *substitution instance.*

Program Equivalence It is important to be able to say when two programs `p, q` $\in D$ are computationally equivalent. In recent years two views have developed, *semantic equivalence* and *operational equivalence.* Both concepts make sense in our framework, as defined by:

Definition 2.5 *Let* `p, q` $\in D$. *Then*

- `p` *and* `q` *are* semantically *equivalent if* $[\![\mathtt{p}]\!] = [\![\mathtt{q}]\!]$
- `p` *and* `q` *are* operationally *equivalent if* $[\![\mathtt{p}]\!] \approx [\![\mathtt{q}]\!]$, *where for $f, g \in V$ we define $f \approx g$ to mean that for all $n \geq 0$ and for all* $\mathtt{d}_1, \ldots, \mathtt{d}_n \in D$ and $\mathtt{d} \in D_\perp$,

$$(f\mathtt{d}_1 \ldots \mathtt{d}_n = \mathtt{d}) \text{ if and only if } (g\mathtt{d}_1 \ldots \mathtt{d}_n = \mathtt{d})$$

The first definition is the simpler of the two, but a case can be made that the second definition is likely to be more relevant in practice. The reason is that the first definition requires verifying equality between two elements of a semantic function domain. This can be a tricky task, and impossible if the semantic function $[\![-]\!]$ is not fully abstract.

The second definition is a version of the more general operational equivalence studied by Plotkin, Milner, and others, limited to first order applicative contexts. It only involves assertions that can in principle be verified by running the program on first order inputs and observing its first order outputs or nontermination behavior.

2.3 Some Problems for Study

1. Find a suitable model theory for these types (domains, ideals, ...). The semantics above uses ordinary sets, but for computational purposes it is desirable that the domains be algebraic, and the values which manipulate programs should only range over computable elements.

$$\begin{tikzcd}
\underline{(\alpha \to \beta)} \times \underline{(\beta \to \gamma)} \arrow[r, "\underline{;}"] \arrow[d, "\llbracket - \rrbracket \times \llbracket - \rrbracket"] & \underline{(\alpha \to \gamma)} \arrow[d, "\llbracket - \rrbracket"] & \text{Syntactic composition} \\
(\alpha \to \beta) \times (\beta \to \gamma) \arrow[r, ";"] & (\alpha \to \gamma) & \text{Semantic composition}
\end{tikzcd}$$

Figure 2: Symbolic Composition

2. Find a sound set of type rules for a familiar language that are sufficiently powerful to describe the types of compilers and interpreters (more on this in section 5.3).
3. Figure 1 contains no rules for deducing that any programs at all have types of form $\underline{t}_{\mathbf{L}}$. Problem: for a fixed programming language, find type inference rules appropriate for showing that given programs have given types. (More will be said on this in a later section.)

3 Efficient Symbolic Composition

Symbolic composition can be described as commutativity of the diagram in Figure 2, where α, β, γ range over all types. We now list several examples of symbolic composition, and discuss what is saved computationally.

3.1 Vector Spaces and Matrix Multiplication

An $m \times n$ matrix M over (for example) the real numbers R determines a linear transformation $\llbracket M \rrbracket : R^n \to R^m$, so $\llbracket M \rrbracket \vec{v}$ is a vector $\vec{w} \in R^n$ if $\vec{v} \in R^m$. If M, N are respectively $m \times n$ and $n \times p$ matrices and $M \cdot N$ their matrix product, then $\llbracket M \cdot N \rrbracket : R^p \to R^m$ and we have

$$\llbracket M \cdot N \rrbracket(\vec{w}) = \llbracket M \rrbracket(\llbracket N \rrbracket(\vec{w}))$$

Assuming $m = n = p$, the composition $\llbracket M \rrbracket(\llbracket N \rrbracket(\vec{w}))$ can be computed in either of two ways:

- by applying first N and then M to $\vec{w}$ (time $2n^2$), or
- by first multiplying M and N (time n^3 by the naive algorithm), and applying the result to $\vec{w}$ (time n^2)

It may be asked: what if anything has been saved? The answer is: nothing, if the goal is only to transform a single vector, since the second time always exceeds the first. There is, however, a net saving if more than n vectors are to be transformed since the matrix multiplication need only be computed once.

The moral: so familiar an operation as matrix multiplication can be thought of as symbolic composition; and composition can save computational time.

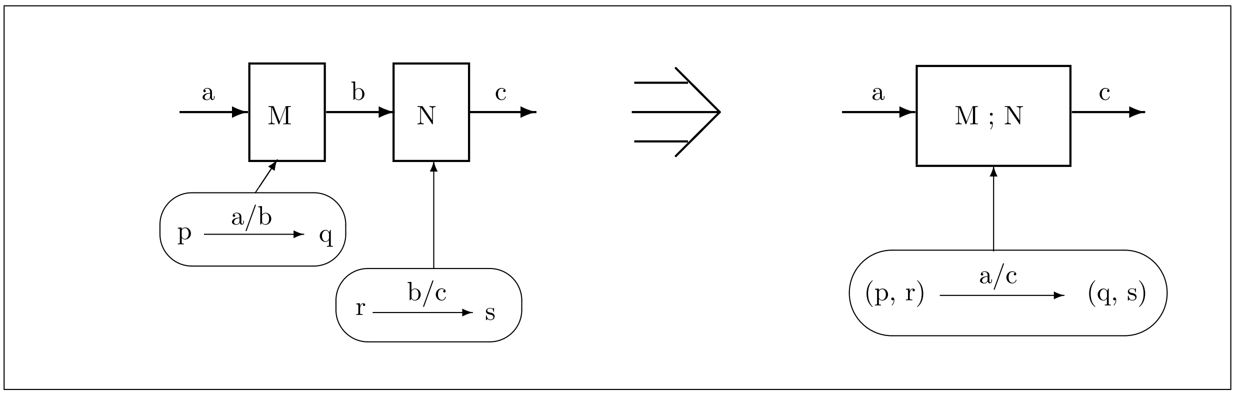


Figure 3: Composition of Finite State Transducers

3.2 Finite State Transducers

Suppose one is given two finite state transducers: M which transforms input strings from Σ^* into Δ^*, and N which transforms strings from Δ^* into Γ^*. It is well known that these can be combined into a single finite state transducer M;N that directly transforms strings from Σ^* into Γ^* without ever producing any intermediate strings. The construction is straightforward: a state of M;N is a pair of states, one from M and one from N. Transitions in M;N are defined as indicated in the diagram of Figure 3.

3.3 Derivors

A derivor (terminology taken from the ADJ group [9]) is a translation from one many-sorted algebra to another, usually involving a change of signature (a signature morphism is an example). Derivors must be *compositional*: the translation of a composite term is a combination of the translations of its components. The requirement of compositionality is very natural for *denotational semantics* [17].

Derivors over term algebras can also be thought of as finite state transducers that have been extended to operate on trees (strings are a special case). Derivors can also be composed symbolically in a way very analogous to finite transducers. For a small example, consider the three term algebras $T(A), T(B), T(C)$ over signatures given in Figure 4.

Using a natural denotational notation we now define two derivors. One is $[\![-]\!]_{AB}$ from $T(A)$ to $T(B)$:

$$\begin{array}{lcl}
[\![0]\!]_{AB} & = & \mathit{zero} \\
[\![1]\!]_{AB} & = & \mathit{one} \\
[\![a_1 + a_2]\!]_{AB} & = & [\![a_1]\!]_{AB}\ \mathit{plus}\ [\![a_2]\!]_{AB} \\
[\![a_1 - a_2]\!]_{AB} & = & [\![a_1]\!]_{AB}\ \mathit{plus}\ (\mathit{minus}\ [\![a_2]\!]_{AB})
\end{array}$$

and the second is $[\![-]\!]_{BC}$ from $T(B)$ to $T(C)$:

$$\begin{array}{lcl}
[\![\mathit{zero}]\!]_{BC} & = & \mathit{push}(0) \\
[\![\mathit{one}]\!]_{BC} & = & \mathit{push}(1) \\
[\![b_1\ \mathit{plus}\ b_2]\!]_{BC} & = & [\![b_1]\!]_{BC}\ ;\ [\![b_2]\!]_{BC};\ \mathit{add} \\
[\![\mathit{minus}\ b]\!]_{BC} & = & [\![b]\!]_{BC}\ ; \mathit{negate}
\end{array}$$

Their composition $[\![-]\!]_{AC}$ from $T(A)$ to $T(C)$ is obtained by applying $[\![-]\!]_{BC}$ to the right side of every rule defining $[\![-]\!]_{AB}$. This process yields a more efficient version in which no B terms are ever constructed:

$$\begin{array}{lcl}
[\![0]\!]_{AC} & = & \mathit{push}(0) \\
[\![1]\!]_{AC} & = & \mathit{push}(1) \\
[\![a_1 + a_2]\!]_{AC} & = & [\![a_1]\!]_{AC}\ ;\ [\![a_2]\!]_{AC};\ \mathit{add} \\
[\![a_1 - a_2]\!]_{AC} & = & [\![a_1]\!]_{AC}\ ;\ ([\![a_2]\!]_{AC}; \mathit{negate});\ \mathit{add}
\end{array}$$

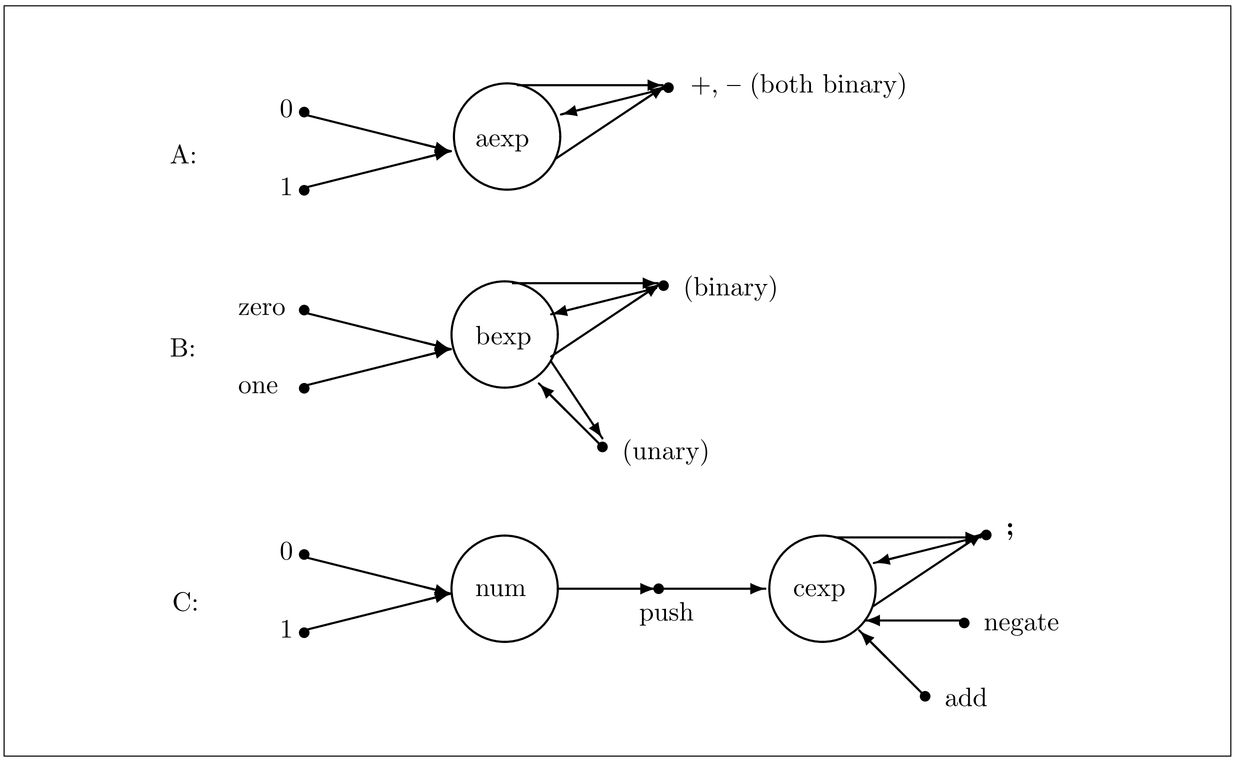


Figure 4: Three Signatures

3.4 Attribute Coupled Grammars

These define term-to-term mappings; typically the terms are abstract syntax trees. Attribute coupled grammars have much in common with derivors, but extend them significantly in several ways. One is that multiple attributes are allowed, and another is that attribute values may flow both up and down a term. (Derivors correspond to the special case of only one synthesized attribute.) Another extension is to allow nontrivial computation and not just the assembly of subtrees.

Ganzinger and Giegerich show in [7] that attribute coupled grammars can be composed symbolically (too complex an operation to illustrate here). Their motivation was to develop a general framework for automatic pass composition and decomposition in the context of multipass compilers.

3.5 Functional programs

We saw an example in the introduction of composing functional programs, with a significant increase in speed. Turchin reports in [20] the transformation of a two pass program into a one pass equivalent. This was followed by Wadler's "listless transformer" [21], more clearly described and for a more traditional lazy functional language. Both approaches are implemented but semiautomatic - they often give good results when they terminate, but are complex and not guaranteed to terminate. Wadler's "treeless transformer" is an advance [22] in that it always terminates, often with significant efficiency improvements, but for a very limited class of programs.

While the state of the art has advanced rapidly, there is still much room for improvement in the symbolic composition of functional programs.

3.6 Some Problems for Study

1. Find a generalization of attribute coupled grammars with a simpler composition algorithm, or with greater computational powers.

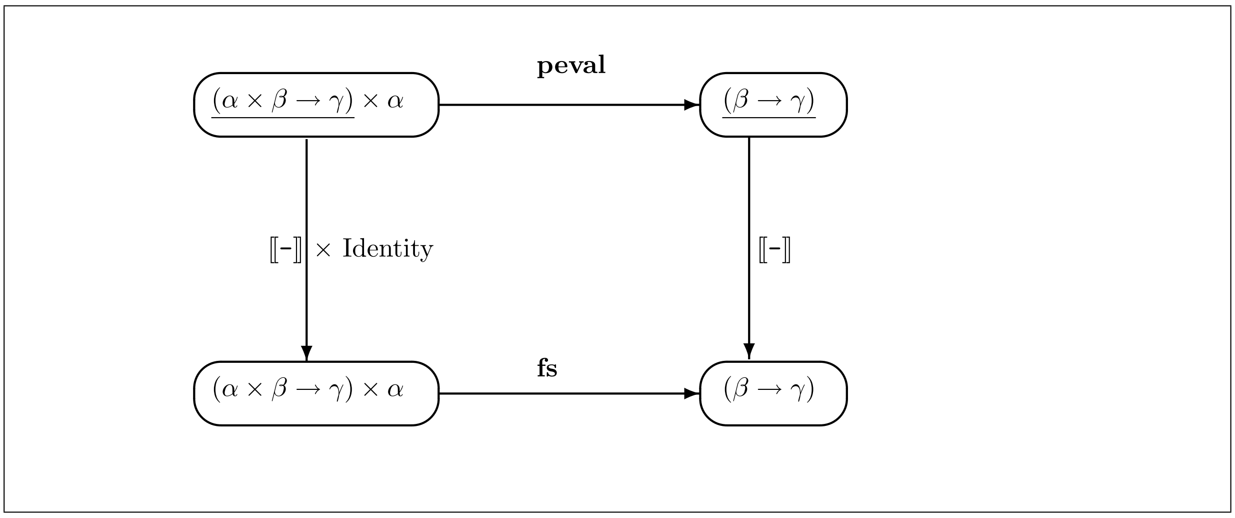

$$\begin{tikzcd}
\underline{(\alpha \times \beta \to \gamma)} \times \alpha \arrow[r, "\textbf{peval}"] \arrow[d, "[\![-]\!] \times \text{Identity}"] & \underline{(\beta \to \gamma)} \arrow[d, "[\![-]\!]"] \\
(\alpha \times \beta \to \gamma) \times \alpha \arrow[r, "\textbf{fs}"] & (\beta \to \gamma)
\end{tikzcd}$$

Figure 5: Function Specialization and Partial Evaluation

2. Find a more general and powerful method for symbolic composition of functional programs.

4 Symbolic Function Specialization = Partial Evaluation

4.1 Function Specialization

The result of *specializing* (also called *restricting*) a two argument function $f(x, y) : \alpha \times \beta \to \gamma$ to a fixed value $x = a$ is the function

$$f|_{x=a}\,(y) = f(a, y)$$

for all y. Function specialization thus has polymorphic type

$$\textbf{fs} : (\alpha \times \beta \to \gamma) \times \alpha \to (\beta \to \gamma)$$

Partial Evaluation Partial evaluation is the symbolic operation corresponding to function specialization. Given a *program* for f and a value $x = a$, a partial evaluator yields a program to compute $f|_{x=a}$. Using **peval** to denote the partial evaluation function, its correctness is expressed by commutativity of the diagram in Figure 5. Partial evaluation has polymorphic type

$$\textbf{peval} : \underline{(\alpha \times \beta \to \gamma)} \times \alpha \to \underline{(\beta \to \gamma)}$$

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 is in essence 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 still other words, a partial evaluator is a computer realization of Kleene's s-m-n theorem [12,16].

The traditional proofs of Kleene's s-m-n theorem do not take efficiency into account, yielding trivial specialized programs. 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.

4.2 Definition of Partial Evaluation

The description of **peval** can be both simplified and generalized by writing the functions involved in *curried* form. (Recall that $\alpha \times \beta{\to}\gamma$ is isomorphic to $\alpha{\to}(\beta{\to}\gamma)$.) This gives **peval** a new polymorphic type:

$$\textbf{peval} : \underline{\alpha\to(\beta\to\gamma)}\to\alpha\to\underline{\beta\to\gamma}$$

which is an instance of the more general polymorphic type:

$$\textbf{peval} : \underline{\rho\to\sigma}\to\rho\to\underline{\sigma}$$

Maintaining our emphasis on observable values, we will require ρ to be a first order type (i.e. *base* or an a type $\underline{t}_{\mathbf{L}}$). A final restriction: **peval** should be computable (as in [12,16]) so we require $\textbf{peval} = [\![\texttt{mix}]\!]$ for some program `mix`. (The name `mix` comes from [5].) Finally, we come to the definition:

Definition 4.1 The `mix` equation. *Program* `mix` $\in D$ *is a* partial evaluator *if for all* `p, a` $\in D$*,*

$$[\![\texttt{p}]\!]\,\texttt{a} \approx [\![[\![\texttt{mix}]\!]\,\texttt{p}\,\texttt{a}]\!]$$

By definition of $\approx$

$$[\![\texttt{p}]\!]\,\texttt{d}_1\texttt{d}_2\ldots\texttt{d}_n = \texttt{d}$$

if and only if

$$[\![[\![\texttt{mix}]\!]\,\texttt{p}\,\texttt{d}_1]\!]\,\texttt{d}_2\ldots\texttt{d}_n = \texttt{d}$$

The same concept may be generalized to differing input, output and implementation languages, but we will only need one-language partial evaluators.

4.3 Some Techniques for Partial Evaluation

Partial evaluators that have been successfully used for compiling and compiler generation include [8], [2], [3] and [11]. The `mix`-time techniques used there include: applying base functions to known data; *unfolding* function calls; and possibly creating one or more *specialized program points.* A program point comes from the source program (oerhaps the name of a user-defined function) and is specialized to known data values computable from the known input $\texttt{d}_1$.

To illustrate these techniques, consider the well-known example of Ackermann's function:

```
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).

By these techniques we can specialize function `a` to the values of `m` = 1 and 2, 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 well illustrates some general optimization techniques.

5 Compilers, Interpreters and Their Types

Let **L**, **S** and **T** be programming languages (respectively source, implementation and target languages). Our default language is **L**, and we often abbreviate $[\![\mathtt{p}]\!]_{\mathbf{L}}$ to $[\![\mathtt{p}]\!]$.

Definition 5.1

1. *An* interpreter for **S** written in **L** *is a member* `int` *of the set*

$$\boxed{\begin{array}{c}\mathbf{S}\\ \mathbf{L}\end{array}} \overset{def}{=} \{\mathtt{int} \in D \mid \forall \mathtt{s} \in D \,.\, [\![\mathtt{s}]\!]_{\mathbf{S}} \approx [\![\mathtt{int}]\!]_{\mathbf{L}}\mathtt{s}\}$$

2. *An* **S**-to-**T**-compiler written in **L** *is a member* `comp` *of the set*

$$\begin{array}{|c|}\hline \mathbf{S} \longrightarrow \mathbf{T} \\ \hline \end{array}\,\boxed{\mathbf{L}} \overset{def}{=} \{\mathtt{comp} \in D \mid \forall \mathtt{s} \in D \,.\, [\![\mathtt{s}]\!]_{\mathbf{S}} \approx [\![[\![\mathtt{comp}]\!]_{\mathbf{L}}\mathtt{s}]\!]_{\mathbf{T}}\}$$

5.1 The Computational Overhead Caused by Interpretation

On the computer, interpreted programs are often observed to run slower than compiled ones. To explain why, let $t_{\mathtt{p}}(\mathtt{d}_1 \ldots \mathtt{d}_n)$ denote the time required to calculate $[\![\mathtt{p}]\!]_{\mathbf{L}}\mathtt{d}_1 \ldots \mathtt{d}_n$. An interpreter's basic cycle is usually 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 its evaluation function; and finally, actions to evaluate the expression's main operator, e.g. to subtract 1 or to look up the current value of a variable. It is very common in practice that the running time of an interpreter `int` on input `(p b)` satisfies

$$a \cdot t_{\mathtt{p}}(\mathtt{d}) \leq t_{\mathtt{int}}(\mathtt{p\ d})$$

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

This is typical of many interpreters in our experience: an interpreted program runs slower than one which is compiled; and the difference is a constant factor, large enough to be worth reducing for practical reasons, and depending on the size of the program being interpreted. Clever use of data structures (hash tables, binary trees, etc) can make a grow slowly as a function of `p`'s size.

5.2 Can an Interpreter be Typed?

Suppose we have an interpreter `up` for language **L**, and written in the same language—in other words a "universal program" or "self-interpreter". Can `up` be proven type correct? By definition `up` must satisfy

$$[\![\mathtt{p}]\!] \approx [\![\mathtt{up}]\!]\ \mathtt{p}$$

for any **L**-program `p`. Consequently as `p` ranges over all **L**-programs, $[\![\mathtt{up}]\!]\,\mathtt{p}$ can take on *any* program-expressible type. A difficult question arises: is it possible to define the type of `up` nontrivially?

This question does not arise at all in classical computability theory since there is only one data type—the natural numbers, and all programs denote functions on them. On the other hand, computer scientists are unwilling to code all data as numbers and so demand programs with varying input, output and data types.

A traditional Computer Science response to this problem has been to write an interpreter in an *untyped* language, e.g. Lisp or Scheme. This has the disadvantage that it is hard to verify that the interpreter correctly implements the type system of its input language (if any). The reason is that there are two classes of possible errors: those caused by errors in the program being interpreted, and those caused by a badly written interpreter. Without a type system it is difficult to distinguish the one class of interpret-time errors from the other.

An alternative approach is to use the types $\underline{t}_{\mathbf{X}}$ as described earlier. Given this notation and a sufficiently strong type system, we conjecture that it is possible to prove that interpreters, compilers and partial evaluators properly deal with the types of their program inputs and outputs.

5.3 Well-typed Language Processors

Given a source **S**-program denoting a value of some type t, an **S**-interpreter should return a value whose type is t. From the same source program, a compiler should yield a target language program whose **T**-denotation is identical to its source program's **S**-denotation. This agrees with daily experience—a compiler is a meaning-preserving program transformation, insensitive to the type of its input program (provided only that it *is* well-typed). Analogous requirements apply to partial evaluators.

Interpreters A well-typed interpreter is required to have many types: one for every possible input program type. Thus to satisfy these definitions we must dispense with type *unicity*, and allow the type of the interpreted program not to be uniquely determined by its syntax.

Compilers Compilers must satisfy an analogous demand. One example: λ`x.x` has type $\underline{t}_{\mathbf{L}} \to \underline{t}_{\mathbf{L}}$ for all types t. It is thus a trivial but well-typed compiler from **L** to **L**.

Partial Evaluators A well-typed partial evaluator can be applied to any program `p` accepting at least one first order input, together with a value $\mathtt{d}_1$ for `p`'s first input. Suppose `p` has type $\underline{\rho \to \sigma}$ where ρ is first order and $\mathtt{d}_1 \in [\![\rho]\!]$. Then $[\![\mathtt{mix}]\!]\,\mathtt{p}\;\mathtt{d}_1$ is a program whose result type is σ, the type of $[\![\mathtt{p}]\!]\mathtt{d}_1$.

Definition 5.2 [5]

1. *Interpreter* `int` $\in \begin{array}{|c|}\hline \mathbf{S} \\ \mathbf{L} \\ \hline\end{array}$ *is* well-typed *if* $[\![\mathtt{int}]\!]_{\mathbf{L}}$ *has type* $\forall\delta\,.\,\underline{\delta}_{\mathbf{S}} \to \delta$.

2. *Compiler* `comp` $\in \begin{array}{|c|}\hline \mathbf{S} \longrightarrow \mathbf{T} \\ \hline \mathbf{L} \\ \hline\end{array}$ *is* well-typed *if* $[\![\mathtt{comp}]\!]_{\mathbf{L}}$ *has type* $\forall\alpha\,.\,\underline{\alpha}_{\mathbf{S}} \to \underline{\alpha}_{\mathbf{T}}$.

3. *A partial evaluator* `mix` *is* well-typed *if* $[\![\mathtt{mix}]\!]$ *has type* $\forall\rho\,.\forall\sigma\,.\;\underline{\rho \to \sigma} \to \rho \to \underline{\sigma}$, *where* ρ *ranges over first order types.*

Remark The definition of a well-typed interpreter assumes that all **S**-types are also **L**-types (at least, observably so). Thus it does not take into account the possibility of encoding **S**-values, as is often seen in computing practice.

5.4 Problems for Study

1. Verify type correctness of a simple interpreter with integer and boolean data
2. Formulate a set of type inference rules for a full language, e.g. the λ-calculus, that is sufficiently general to verify type correctness of a range of compilers, interpreters and partial evaluators

6 Partial Evaluation and Compiler Generation

6.1 The Futamura Projections

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** produces an equivalent **L**-program `target`, meaning that $[\![\mathtt{source}]\!]_{\mathbf{S}}$ and $[\![\mathtt{target}]\!]_{\mathbf{T}}$ are the same value in V. Recall the `mix` equation:

$$[\![\mathtt{p}]\!]\;\mathtt{a} \approx [\![\,[\![\mathtt{mix}]\!]\;\mathtt{p}\;\mathtt{a}]\!]$$

for all `p, a` $\in B$. The following three equations are the so-called "Futamura projections" [6,5]. They assert that given a partial evaluator `mix` and an interpreter `int`, one may compile programs, generate compilers and even generate a compiler generator.

[5] To say, for example, that $[\![\mathtt{int}]\!]_{\mathbf{L}}$ has type $\forall\delta\,.\,\underline{\delta}_{\mathbf{S}} \to \delta$ means that $[\![\mathtt{int}]\!]_{\mathbf{L}}$ has type $\underline{t}_{\mathbf{S}} \to t$ for every type t.

$$\begin{array}{lcll} [\![\texttt{mix}]\!]\;\texttt{int source} & = & \texttt{target} & \\ [\![\texttt{mix}]\!]\;\texttt{mix int} & = & \texttt{compiler} & \\ [\![\texttt{mix}]\!]\;\texttt{mix mix} & = & \texttt{cogen} & \text{a compiler generator} \end{array}$$

6.1.1 Compilation

Program `source` from the interpreted language **S** has been translated to program `target`. It is natural that `target` is in language **L** since it 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:

$$\begin{array}{lcll} [\![\texttt{source}]\!]_{\mathbf{S}} & \approx & [\![\texttt{int}]\!]\;\texttt{source} & \text{definition of an interpreter} \\ & \approx & [\![[\![\texttt{mix}]\!]\;\texttt{int source}]\!]_{\mathbf{T}} & \texttt{mix}\text{ equation} \\ & = & [\![\texttt{target}]\!]_{\mathbf{T}} & \text{definition of } \texttt{target} \end{array}$$

Why bother? The reason is that it is usually considerably faster to run `target` directly on a **T**-machine than it would be to run `source` on an **L**-machine, due to the removal of the interpretive overhead discussed in section 5.1.

6.1.2 Compiler Generation

We have just seen that `mix` can be used to compile from one language to another, given an interpreter for the source language. The second Futamura projection says that it can also generate stand-alone compilers. Verification that program `compiler` translates source programs into equivalent target programs is also straightforward:

$$\begin{array}{lcll} \texttt{target} & \approx & [\![\texttt{mix}]\!]\;\texttt{int source} & \text{definition of } \texttt{target} \\ & \approx & [\![[\![\texttt{mix}]\!]\;\texttt{mix int}]\!]\texttt{source} & \texttt{mix}\text{ equation} \\ & = & [\![\texttt{compiler}]\!]\;\texttt{source} & \text{definition of a compiler} \end{array}$$

6.1.3 Generating a Compiler Generator

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

$$\begin{array}{lcll} \texttt{compiler} & \approx & [\![\texttt{mix}]\!]\;\texttt{mix int} & \text{definition of } \texttt{target} \\ & \approx & [\![[\![\texttt{mix}]\!]\;\texttt{mix mix}]\!]\;\texttt{int} & \texttt{mix}\text{ equation} \\ & = & [\![\texttt{cogen}]\!]\;\texttt{int} & \text{definition of } \texttt{cogen} \end{array}$$

6.2 Self-Application and Types

Definitions involving self-application often (and rightly) cause concern as to their well-typedness. We show here that natural types for `target` and `compiler` can be deduced from the few type rules given earlier (and even `cogen`, which we omit because of the deduction's complexity). Recall their polymorphic types:

$$\begin{array}{lll} \text{Type of } \texttt{source}: & \underline{\delta}_{\mathbf{S}} & \\ \text{Type of } [\![\texttt{int}]\!]: & \forall\delta\,.\,\underline{\delta}_{\mathbf{S}} \to \delta & \\ \text{Type of } [\![\texttt{compiler}]\!]: & \forall\delta\,.\,\underline{\delta}_{\mathbf{S}} \to \underline{\delta}_{\mathbf{T}} & \\ \text{Type of } [\![\texttt{mix}]\!]: & \forall\rho\,.\,\forall\sigma\,.\,\underline{\rho \to \sigma} \to \rho \to \underline{\sigma} & \text{where } \rho \text{ is first order} \end{array}$$

First Futamura Projection

We wish to find the type of `target` = $[\![\texttt{mix}]\!]$ `int source`. The following deduction assumes that program `source` has type $\underline{\delta}_{\mathbf{S}}$ and concludes that the target program has type $\underline{\delta} = \underline{\delta}_{\mathbf{L}}$, i.e. that it is an **L** program of the right type. The deduction uses only the rules of Figure 1 and generalization of polymorphic variables.

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{mix} : \underline{\underline{\rho\to\sigma}\to\rho\to\underline{\sigma}}}{\llbracket\texttt{mix}\rrbracket : \underline{\rho\to\sigma}\to\rho\to\underline{\sigma}}}{\llbracket\texttt{mix}\rrbracket : \underline{\underline{\delta}_{\mathbf{S}}\to\delta}\to\underline{\delta}_{\mathbf{S}}\to\underline{\delta}} \qquad \texttt{int} : \underline{\underline{\delta}_{\mathbf{S}}\to\delta}}{\llbracket\texttt{mix}\rrbracket\,\texttt{int} : \underline{\delta}_{\mathbf{S}}\to\underline{\delta}} \qquad \texttt{source} : \underline{\delta}_{\mathbf{S}}}{\llbracket\texttt{mix}\rrbracket\,\texttt{int}\,\texttt{source} : \underline{\delta}}$$

Second Futamura Projection

The previous deduction showed that $\llbracket\texttt{mix}\rrbracket\,\texttt{int}$ has the type of a compiling *function*, though it is not a compiler *program*. We now wish to find the type of $\texttt{compiler} = \llbracket\texttt{mix}\rrbracket\,\texttt{mix}\,\texttt{int}$. It turns out to be notationally simpler to begin with a program p of more general type $\underline{\alpha\to\beta}$ than that of int.

$$\dfrac{\dfrac{\dfrac{\dfrac{\texttt{mix} : \underline{\underline{\rho\to\sigma}\to\rho\to\underline{\sigma}}}{\llbracket\texttt{mix}\rrbracket : \underline{\rho\to\sigma}\to\rho\to\underline{\sigma}}}{\llbracket\texttt{mix}\rrbracket : \underline{\underline{\alpha\to\beta}\to\alpha\to\underline{\beta}}\to\underline{\alpha\to\beta}\to\underline{\alpha\to\underline{\beta}}} \qquad \texttt{mix} : \underline{\underline{\alpha\to\beta}\to\alpha\to\underline{\beta}}}{\llbracket\texttt{mix}\rrbracket\,\texttt{mix} : \underline{\alpha\to\beta}\to\underline{\alpha\to\underline{\beta}}} \qquad \texttt{p} : \underline{\alpha\to\beta}}{\llbracket\texttt{mix}\rrbracket\,\texttt{mix}\,\texttt{p} : \underline{\alpha\to\underline{\beta}}}$$

Some Interesting Substitution Instances

By the second Futamura projection, $\texttt{compiler} = \llbracket\texttt{mix}\rrbracket\,\texttt{mix}\,\texttt{int}$. The type of $\texttt{p} = \texttt{int}$ is $\underline{\underline{\delta}_{\mathbf{S}}\to\delta}$, an instance of the type assigned to p above. By the same substitution we have $\llbracket\texttt{compiler}\rrbracket : \underline{\delta}_{\mathbf{S}}\to\underline{\delta}$. Furthermore δ was chosen arbitrarily, so $\llbracket\texttt{compiler}\rrbracket : \forall\delta\,.\,\underline{\delta}_{\mathbf{S}}\to\underline{\delta}$ as desired.

The Type of a Compiler Generator. Even cogen can be given a type, namely

$$\underline{\underline{\alpha\to\beta}\to\underline{\underline{\alpha\to\underline{\beta}}}}$$

by exactly the same technique; but the tree is too complex to display. One substitution instance of cogen's type is the conversion of an interpreter's type into that of a compiler.

The type of $\llbracket\texttt{cogen}\rrbracket$ is clearly $\underline{\alpha\to\beta}\to\underline{\alpha\to\underline{\beta}}$. This is just an instance of the type of the identity function(!) but with some underlining. It is substantially different, however, in that it describes program generation. Specifically

1. $\llbracket\texttt{cogen}\rrbracket$ transforms a two input program p into another program p-gen, such that for any $\texttt{d}_1 \in D$
2. $\texttt{p'} = \llbracket\texttt{p-gen}\rrbracket\texttt{d}_1$ is a program
3. which for any $\texttt{d}_2 \in D$ computes
$$\llbracket\texttt{p'}\rrbracket\texttt{d}_2 \approx \llbracket\texttt{p}\rrbracket\texttt{d}_1\,\texttt{d}_2$$

Ershov called program `p-gen` the *generating extension* of `p`—a logical name since it is a program that generates specialized versions of `p`, when given values $\mathtt{d}_1$ of its first input.

One could even argue that the function $[\![\texttt{cogen}]\!]$ realizes an *intensional version of currying*, one that works on program texts instead of on functions[6]. To follow this, the type of $[\![\texttt{cogen}]\!]$ has as a substitution instance

$$[\![\texttt{cogen}]\!] : \underline{\alpha\to(\beta\to\gamma)}\to\underline{\alpha\to\underline{\underline{\beta\to\gamma}}}$$

In most higher order languages it requires only a trivial modification of a program text with type $(\alpha\to(\beta\to\gamma))$ to obtain a variant with type $(\alpha\times\beta\to\gamma)$, and with identical computational complexity. So a variant `cogen'` could be easily constructed that would first carry out this modification on its program input, and then run `cogen` on the result. The function computed by `cogen'` would be of type:

$$[\![\texttt{cogen'}]\!] : \underline{\alpha\times\beta\to\gamma}\to\underline{\alpha\to\underline{\underline{\beta\to\gamma}}}$$

6.3 Efficiency Issues

6.3.1 Self-Application can Improve Efficiency

A variety of partial evaluators satisfying all the above equations have been constructed ([8,3,11] contain more detailed discussions). Compilation, compiler generation and compiler generator generation can each be done in two ways, to wit:

$$\begin{array}{lcl}
\texttt{target} & = & [\![\texttt{mix}]\!]\,\texttt{int source} \\
 & = & [\![\texttt{compiler}]\!]\,\texttt{source} \\
 & & \\
\texttt{compiler} & = & [\![\texttt{mix}]\!]\,\texttt{mix int} \\
 & = & [\![\texttt{cogen}]\!]\texttt{int} \\
 & & \\
\texttt{cogen} & = & [\![\texttt{mix}]\!]\,\texttt{mix mix} \\
 & = & [\![\texttt{cogen}]\!]\,\texttt{mix}
\end{array}$$

Although the exact timings vary according to the partial evaluator and the implementation language **L**, in each case the second way is often about 10 times faster than the first. A less machine dependent and more intrinsic efficiency measure follows.

6.3.2 Optimality of 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.

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 a self-interpreter

$$\texttt{up} \in \boxed{\begin{array}{c}\mathbf{L}\\ \mathbf{L}\end{array}}$$

For any program `p`

$$[\![\texttt{p}]\!] \quad\approx\quad [\![\texttt{up}]\!]\,\texttt{p} \quad\approx\quad [\![[\![\texttt{mix}]\!]\,\texttt{up p}]\!]$$

so $\texttt{p}' = [\![\texttt{mix}]\!]\,\texttt{up p}$ 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`.

[6]The well-known "curry" isomorphism on functions is:

$$\mathit{curry} : (\alpha\to(\beta\to\gamma)) \simeq (\alpha\times\beta\to\gamma)$$

$$\begin{tikzcd}
\text{Source syntax} \arrow[r, "\text{compile}"] \arrow[d, "[\![-]\!]_{Source}"] & \text{Target syntax} \arrow[d, "[\![-]\!]_{Target}"] \\
\text{Source meanings} \arrow[r, dashed, shift left, "\text{encode}"] & \text{Target meanings} \arrow[l, dashed, shift left, "\text{decode}"]
\end{tikzcd}$$

Figure 6: Morris and ADJ Advice on Compiler Construction

Definition 6.1 `Mix` *is* optimal *provided* $\texttt{p} =_\alpha [\![\texttt{mix}]\!]\ \texttt{up}\ \texttt{p}$ *for all* $\texttt{p} \in B$.

We have satisfied this criterion on the computer for several partial evaluators for various self-interpreters (e.g. [11]). In each case the residual program is identical to `p` (up to variable renaming).

7 Symbolic Composition and Compiler Generation

We describe two algebraic approaches to compiler generation in which symbolic composition plays a central role.

7.1 The Morris and ADJ Advice on Compiler Construction

This does not involve symbolic composition, but we mention it briefly to put the next two examples into perspective. The first "advice" by Morris [13] was to express the translation algebraically, regarding source program syntax, target program syntax, and source and target meanings as many-sorted algebras, and regarding both the semantic and compilation functions as homomorphisms. Derivors were used for the necessary signature changes.

An important consequence is that compiler correctness can be proven by purely algebraic means. The ADJ group [10] carried Morris' advice further, extending his results to a more powerful source language and providing a more purely algebraic correctness proof. Correctness was proven by showing commutativity of versions of the diagram in Figure 6.

The Morris/ADJ algebraic approaches provide a rich framework for constructing and manipulating programs, program pieces and their meanings, and does indeed provide a way to prove compilers correct. However their advice is less suitable if the goal is *compiler generation* for a variety of source languages. One problem is that it requires a large amount of work to define the appropriate algebras and semantic functions - work that must be redone from scratch for every new language or compiler. Further, devising "encode" or "decode" functions that work is a decidedly nontrivial task, and one hard to do systematically.

7.2 Denotational Semantics

Denotational semantics provides a systematic framework for defining semantics of a wide range of programming languages. A denotational language definition specifies program phrase meanings in two steps with the aid of a single, universal "semantic language" which we will call **Sem** (quite often the λ-calculus[7]). The effect is that a program's meaning is *by definition* the meaning of the semantic language expression into which it is mapped. Clearly **Sem** should be a broad spectrum language, suitable for assigning meanings to a wide variety of programming languages.

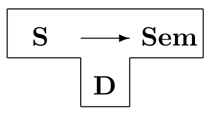
The classical "denotational assumption" is just compositionality, so the mapping from source language terms can be given by a derivor from the source syntax algebra **S** to the semantic algebra **Sem**. To describe this more formally, let **D** be the language of derivors, so derivor `def` defines a mapping $[\![\texttt{def}]\!]_{\mathbf{D}}$ from source language terms to semantic language terms. The two step nature is seen in the following diagram:

$$\begin{tikzcd}
\boxed{\text{Source algebra } \mathbf{S}} \arrow[r, "{[\![\texttt{def}]\!]_{\mathbf{D}}}"] & \boxed{\text{Semantic algebra } \mathbf{Sem}} \arrow[r, "{[\![-]\!]_{\mathbf{Sem}}}"] & \boxed{\text{Program meanings}}
\end{tikzcd}$$

where for any **S**-program `s`

$$[\![\texttt{s}]\!]_{\mathbf{S}} = [\![[\![\texttt{def}]\!]_{\mathbf{D}}\texttt{s}]\!]_{\mathbf{Sem}}$$

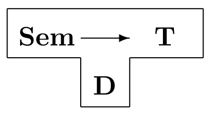
by definition. An equivalent statement: definition `def` is a compiler from **S** to **Sem**(!). Expressed symbolically we have

$\texttt{def} \in$ 

7.3 Mosses' "Constructive Approach to Compiler Correctness"

The Morris/ADJ diagram can be simplified if we demand source and target meanings to be identical, and we do so in this and the next section. With this convention, Figure 7 illustrates Mosses' approach, where **T** is the target term algebra and $[\![-]\!]_{\mathbf{T}}$ is its semantic function.

One of Mosses' ideas was to implement the semantic language once and for all by means of a derivor, and to *re-use* this fixed implementation to generate compilers for a variety of source languages. The essential property of the implementation can be described by:

$\texttt{imp} \in$ 

We said before that derivors can be composed symbolically, a fact we now state more formally using the "tee diagram" notation (proof is immediate):

[7]While λ-calculus is often used, it suffers a number of practical disadvantages which led Mosses to develop an alternative semantic framework called "action semantics". The choice of semantic language is not relevant to our discussion, so we say no more about this otherwise interesting subject.

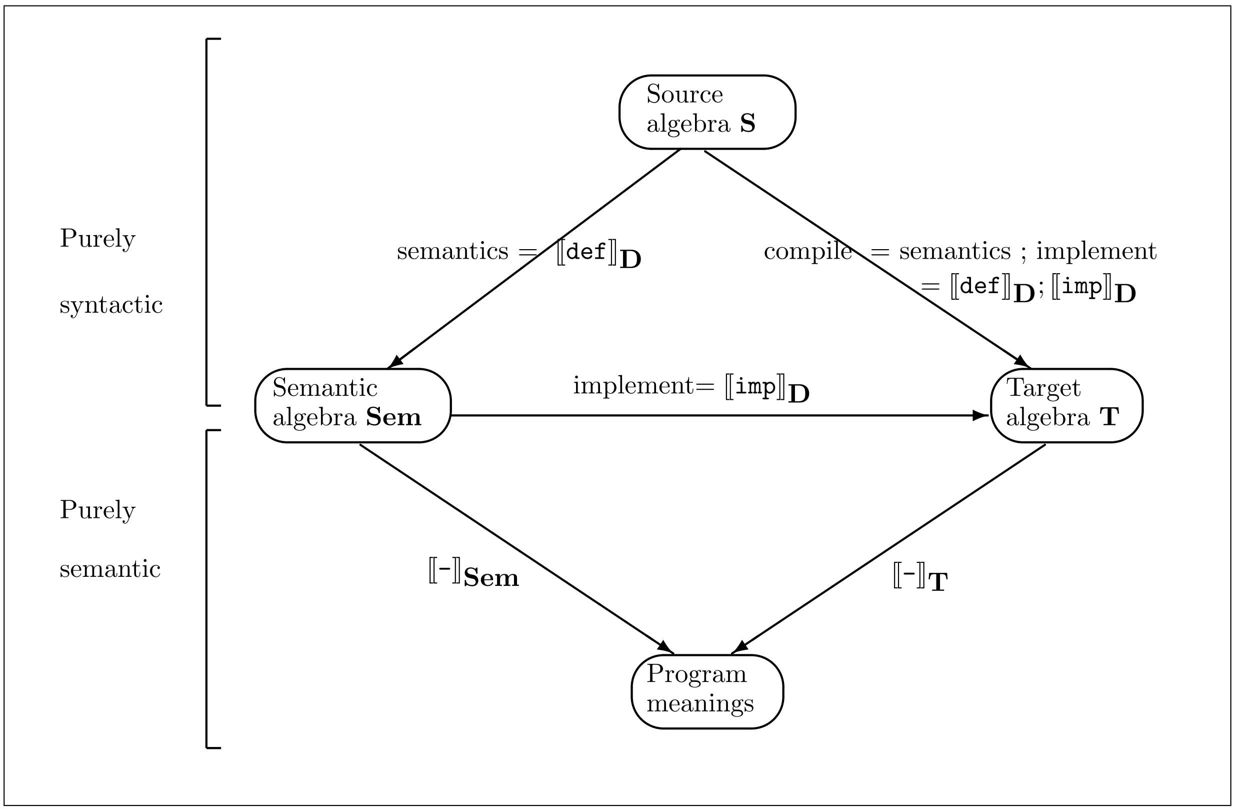
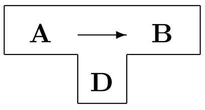
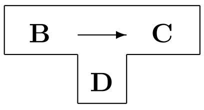
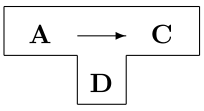


Figure 7: Mosses' Approach to Compiler Construction

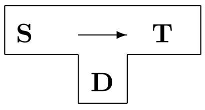
Proposition 7.1 *If* `d` ∈ 

and `p` ∈ 

then

`d` $\underset{-}{;}$ `p` ∈ 

A consequence is that compilers for various source languages may be obtained by symbolically composing their source semantics with the fixed implementation derivor `imp`. By the proposition just stated `comp` = `def` $\underset{-}{;}$ `imp` is a correct compiler from **S** to **T**, in the form of a derivor:

`comp` = `def` $\underset{-}{;}$ `imp` ∈ 

Correctness of this compiler generation scheme is ensured by commutativity of the diagram. Note that it is only necessary to establish correctness of the lower triangle; then any source semantics whatever will be converted into a correct compiler by symbolic composition.

7.4 CERES

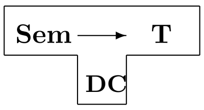
Mosses' approach is definitely a more general method for compiler generation than the Morris/ADJ technique. On the other hand, the constructed compiler is still a derivor, so compilation can only be done homomorphically. This limits the range of possible target programs, for instance identical subterms of a program will always be translated identically. Further, while derivors realize syntax-directed translations, they cannot express iterations that are unbounded according to source program structure, but such techniques are frequently seen in realistic compiler optimizations.

The starting point of the CERES approach to compiler generation [18] is to introduce a separate algebra **C** of *compile time actions*—a language appropriate for expressing compiler operations. As before the universal implementation `imp` is a derivor. However the diagram is changed so `imp` maps semantic expressions into "compile time actions" in **C**. The net effect is that compilation is not achieved by a derivor in one step, as with Mosses' approach, but in two steps (conceptually at least).

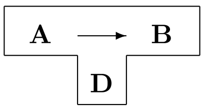
Given a an **Sem**-term `p`, the first step is computation of $[\![\mathtt{int}]\!]_{\mathbf{D}}\ \mathtt{p}$. The result is a compile time action—a term in algebra **C**. The second step is evaluation of that compile time action via $[\![\mathtt{-}]\!]_{\mathbf{C}}$. We invent a new language **DC** to describe this process, with definition:

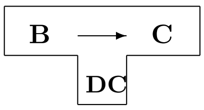
$$[\![\mathtt{def}]\!]_{\mathbf{DC}}\mathtt{p} = [\![\,[\![\mathtt{def}]\!]_{\mathbf{D}}\,\mathtt{p}]\!]_{\mathbf{C}}$$

Given this, the type of `imp` can alternatively (and equivalently) be described by a tee diagram analogous to that of Mosses' method:

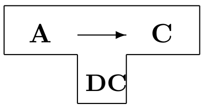
`imp` $\in$ 

As by Mosses' method, compiler construction is done by symbolically composing `def` with `imp`. The composition of `def` with `imp` is thus not a target program itself, but rather maps source **S**-programs to compile time actions which, when performed, construct the target program. A result analogous to the proposition above is easy to verify:

Proposition 7.2 *If* `d` $\in$ 

and `p` $\in$ 

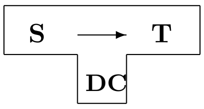
then

$\mathtt{d}\ \underline{;}\ \mathtt{p}$ $\in$ 

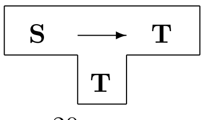
The net effect is to make possible more sophisticated compilation schemes than derivors alone can accomplish. Correctness of the scheme is expressed by commutativity of the diagram of Figure 8.

Producing Compilers in Target Code

The compilers resulting from this scheme are in language **DC**:

$\mathtt{def}\ \underline{;}\ \mathtt{imp}$ $\in$ 

It is of course desirable to obtain a compiler in target language form:



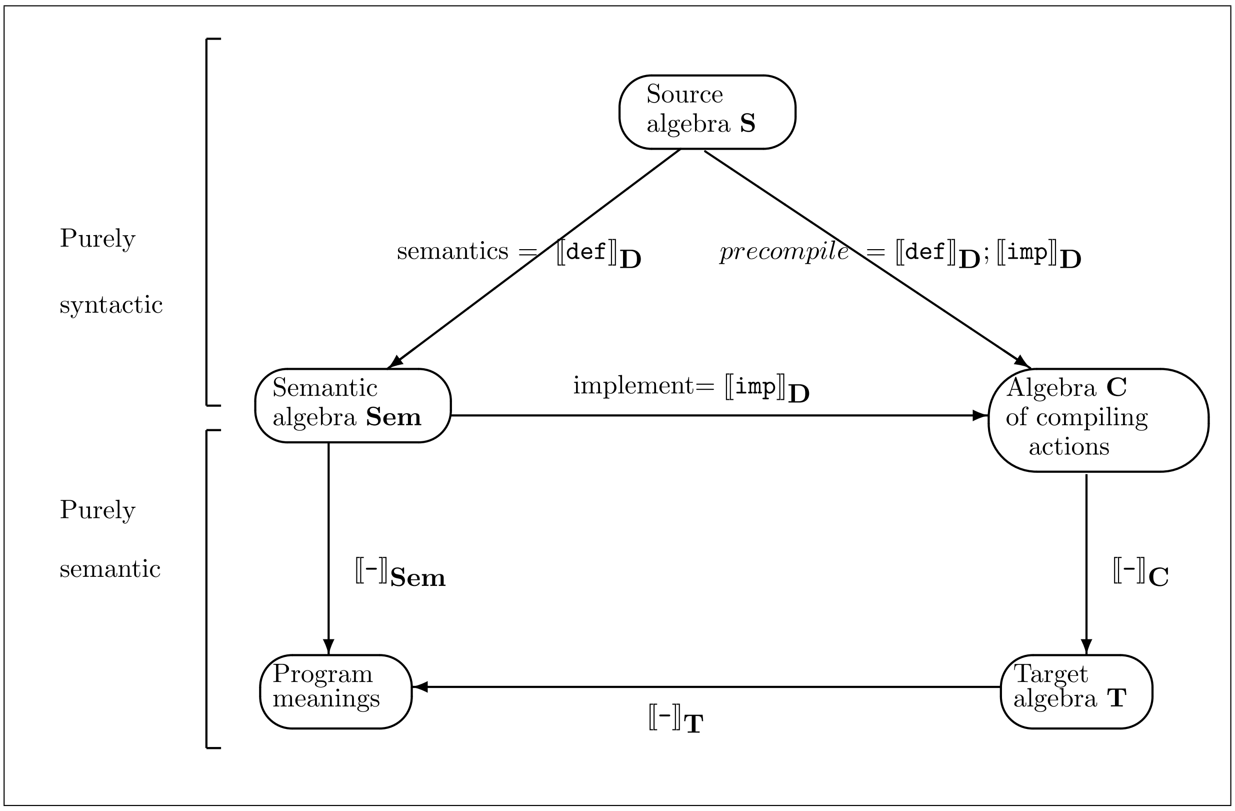

Purely syntactic
Purely semantic
Source algebra **S**
semantics = $[\![\mathtt{def}]\!]_{\mathbf{D}}$
precompile = $[\![\mathtt{def}]\!]_{\mathbf{D}}; [\![\mathtt{imp}]\!]_{\mathbf{D}}$
Semantic algebra **Sem**
implement= $[\![\mathtt{imp}]\!]_{\mathbf{D}}$
Algebra **C** of compiling actions
$[\![\text{-}]\!]_{\mathbf{Sem}}$
$[\![\text{-}]\!]_{\mathbf{C}}$
Program meanings
Target algebra **T**
$[\![\text{-}]\!]_{\mathbf{T}}$

Figure 8: CERES Approach to Compiler Construction

This can be done with the aid of a bootstrapping process described in [18]. The technique is too involved to explain here, but is based on yet another example of symbolic computation called *self-composition* which we just mention in the next subsection. Surprisingly (and in spite of the two-step definition of language **DC**), the generated compilers have only one pass.

Yet Another Application of Symbolic Computation

The fundamental observation is that the essence of this approach to compiler generation is the transformation

$$\mathtt{def} \Rightarrow \mathtt{def} \underset{\cdot}{;} \mathtt{imp}$$

This transformation will be performed for many user-written semantic definitions `def`, but all with the same implementation `imp`. Tofte shows in [18] that for any derivor `imp` there exists another derivor `imp'` with the property that for any derivor `def`,

$$[\![\mathtt{imp'}]\!]_{\mathbf{D}}\mathtt{def} = \mathtt{def} \underset{\cdot}{;} \mathtt{imp}$$

In words, `imp'` accepts derivor `def` as its input data value, and yields as output the result of composing `def` with `imp`. A detailed explanation of how this leads to a practically useful technique for bootstrapping (as it does!) is beyond the scope of this article.

7.5 Problems for Study

1. Experiment with the CERES techniques using a stronger definition language , e.g. attribute-coupled grammars or a functional language.

2. Mosses' and CERES' techniques allow automation of code generation, but do not naturally handle compile-time computations, e.g. constant propagation, mapping variables to (base, offset) pairs, etc. Problem: extend the framework to allow a broader range of traditional compiling techniques.

8 Conclusions

Symbolic operations on programs have been seen to be widespread, and useful in several ways for compilation. A type system was developed that seems quite suitable for describing compilers, interpreters and other language processors. It was shown to give sensible types even in the case of the self-application used for generating compilers from interpreters via the Futamura projections. Further, the tee and box notations for compilers and interpreters have shown their descriptive powers.

A number of open problems have been stated along the way.

A Final Problem for Study is to develop a truly general framework for symbolic computation on programs—one in which symbolic composition, partial evaluation and other as-yet-unthought-of transformations can be expressed. A natural candidate would seem to be *Cartesian categorical combinators*, due their generality, high level (e.g. composition and currying are primitives), their proven utility in compiling (i.e. the Categorical Abstract Machine [4]), and the many algebraic laws that can be used to manipulate them.

References

[1] Bjørner, D., A. P. Ershov, N. D. Jones, *Proc. Workshop on Partial Evaluation and Mixed Computation,* North-Holland, 1988

[2] A. Bondorf, O. Danvy: Automatic autoprojection of recursive equations with global variables and abstract data types. Accepted by Science of Computer Programming, 1991.

[3] A. Bondorf: Automatic autoprojection of higher order recursive equations. ESOP Proceedings, Lecture Notes in Computer Science 432, 1990, Springer-Verlag. Expanded version accepted by Science of Computer Programming, 1991.

[4] G. Cousineau: The categorical abstract machine. Chapter 3, pp. 25-45, *University of Texas Year of Programming Series*, Addsion-Wesley, 1990.

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

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

[7] H. Ganzinger, R. Giegerich: Attribute-coupled grammars. Proc. ACM Sigplan Symposium on Compiler Construction, Montreal, Canada, 1984 .

[8] N. D. Jones, Peter Sestoft, Harald Søndergaard: MIX: A Self-applicable Partial Evaluator for Experiments in Compiler Generation. Lisp and Symbolic Computation, vol. 2, no. 1, pp. 9-50, 1989.

[9] J. Goguen, J. Thatcher, E. Wagner: An initial Algebra approach to the specification, correctness and implementation of abstract data types. In **Current trends in programming Methodology IV** (ed. R. T. Yeh), pp. 80-149, Prentice-Hall, 1978

[10] J. Goguen, J. Thatcher, E. Wagner: More on advice on structuring compilers and proving them correct. In *Semantics-Directed Compiler Generation*, pp. 165-188, Springer-Verlag 94, 1980.

[11] 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, 1990.

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

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

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

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

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

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

[18] Tofte, Mads, *Compiler generators - What they can do, what they might do and what they will probably never do*, 146 pp., EATCS Monographs, Springer-Verlag, 1990

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

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

[21] P. Wadler: Listlessness Is Better than Laziness: Lazy Evaluation and Garbage Collection at Compile-time. ACM Symposium on LISP and Functional Programming, Austin, Texas, pp 45-52, 1984.

[22] P. Wadler: Deforestation: Transforming Programs to Eliminate Trees. Proceedings of the Workshop on Partial Evaluation and Mixed Computation, North-Holland, 1988.

Contents

1 Introduction **1**
1.1 High-level Operations in Programming Languages . . . 1
1.2 Operations on Functions and Programs . . . 2
1.3 Efficient Operations on Programs . . . 2
1.4 Program Running Times . . . 3
1.5 Goals of this Article . . . 3

2 Definitions and Terminology **4**
2.1 Programming Languages . . . 5
2.2 Data and Program Types . . . 5
2.3 Some Problems for Study . . . 6

3 Efficient Symbolic Composition **7**
3.1 Vector Spaces and Matrix Multiplication . . . 7
3.2 Finite State Transducers . . . 8
3.3 Derivors . . . 8
3.4 Attribute Coupled Grammars . . . 9
3.5 Functional programs . . . 9
3.6 Some Problems for Study . . . 9

4 Symbolic Function Specialization = Partial Evaluation **10**
4.1 Function Specialization . . . 10
4.2 Definition of Partial Evaluation . . . 10
4.3 Some Techniques for Partial Evaluation . . . 11

5 Compilers, Interpreters and Their Types **12**
5.1 The Computational Overhead Caused by Interpretation . . . 12
5.2 Can an Interpreter be Typed? . . . 12
5.3 Well-typed Language Processors . . . 13
5.4 Problems for Study . . . 13

6 Partial Evaluation and Compiler Generation **13**
6.1 The Futamura Projections . . . 13
6.1.1 Compilation . . . 14
6.1.2 Compiler Generation . . . 14
6.1.3 Generating a Compiler Generator . . . 14
6.2 Self-Application and Types . . . 14
6.3 Efficiency Issues . . . 16
6.3.1 Self-Application can Improve Efficiency . . . 16
6.3.2 Optimality of Partial Evaluation . . . 16

7 Symbolic Composition and Compiler Generation **17**
7.1 The Morris and ADJ Advice on Compiler Construction . . . 17
7.2 Denotational Semantics . . . 17
7.3 Mosses' "Constructive Approach to Compiler Correctness" . . . 18
7.4 CERES . . . 19
7.5 Problems for Study . . . 21

8 Conclusions **22**