The Register-Closure Abstract Machine: A Machine Model for CPS Compiling

Mitchell Wand [*]

College of Computer Science
Northeastern University
360 Huntington Avenue, 161CN
Boston, MA 02115, USA

September 6, 1989

Abstract

We present a new abstract machine model for compiling languages based on their denotational semantics. In this model, the output of the compiler is a λ-term which is the higher-order abstract syntax for an assembly language program. The machine operates by reducing these terms. This approach is well-suited for generating code for modern machines with many registers. We discuss how this approach can be used to prove the correctness of compilers, and how it improves on our previous work in this area.

[*]Work supported by the National Science Foundation under grants numbered DCR-8605218 and CCR-8801591.

1 Introduction

The idea of using conversion to continuation-passing style as the core of compiling is an old one. In previous work, the source language (typically λ-calculus-like) was converted to cps by a cps-conversion algorithm. One then observed that cps code was similar to a flowchart, so one could compile the resulting flowchart by conventional means [7].

In this paper we take a different approach. We use denotational semantics to generate λ-terms in continuation-passing style. We observe that these terms constitute the higher-order abstract syntax of an assembly language; ordinary assembly language can be obtained by using a suitable concrete syntax.

We then construct a machine model (the *Register-Closure Abstract Machine*) for which our "higher-order abstract assembly language" is the machine language. We show how the behavior of this machine can be derived from the definitions of the combinators from which the terms are built. This machine model is the key technical development of this paper. It provides a formal basis for the connection between cps code and machine code.

The resulting approach to compiling is very flexible, because it allows the compiler writer to specify a greater range of machine instructions in the model. Rather than constraining the compiler writer to a particular methodology, it provides a correctness criterion which can be applied to arbitrarily clever compiler designs.

In this paper, we first introduce the idea of higher-order abstract syntax and illustrate how it works for some simple assembly-language constructs. In Section 3, we develop the machine model. Then, in Section 4, we show how this apparatus can be used to develop and prove the correctness of a tiny compiler. In Section 5, we show how to extend our machine and compiler to handle procedures. The resulting language is approximately the same as that in [15]. Last, in Section 6, we discuss previous work and extensions.

2 The Higher-Order Abstract Syntax of Assembly Language

A higher-order abstract syntax [9], also sometimes called Church encoding [4], is a representation of the abstract syntax of a phrase as a λ-term in which the binding relationships in the source phrase are represented using the binding relationships in the λ-term. For example, a quantified formula $\forall x.A$ in first-order logic might be represented as $\forall(\lambda x.A)$. This shows how

x is bound in A.

Let us see how this idea works for assembly language. Consider first a simple 3-register addition instruction **add** r_1, r_2, r_3, which puts the sum of the contents of registers r_1 and r_2 in register r_3, and goes on to the next instruction. We would like to create a higher-order abstract syntax for this bit of concrete syntax. We observe that we must take control flow into account, so we consider instead the instruction stream **add** $r_1, r_2, r_3; C$ beginning with our addition instruction.

The purpose of higher-order abstract syntax is to represent the binding patterns in the syntax, before semantics is considered. It is clear that r_1 and r_2 are "free" in this sequence, because the meaning of the sequence, whatever it is, depends on their values. The register r_3, however, is not free, since it is set rather than referenced. In fact, the addition instruction *binds* r_3 for the remainder of the sequence C. Therefore we define the abstract syntax of **add** $r_1, r_2, r_3; C$ to be

$$\mathbf{add}\, r_1\, r_2\, (\lambda r_3.C)$$

for some combinator **add**.

Now, given this syntax, it is reasonably clear that the definition of **add** ought to be:

$$\mathbf{add} = \lambda x y \kappa . \kappa(x + y)$$

Similarly, the abstract syntax for a **move** instruction, whose concrete syntax is **move** $r_1, r_2; C$ should be

$$\mathbf{move}\, r_1(\lambda r_2.C)$$

and its semantics should be

$$\mathbf{move} = \lambda x \kappa . \kappa x$$

Last, consider a **halt** instruction. Its abstract syntax is

$$\mathbf{halt}\, r$$

and its semantics is $\mathbf{halt} = \lambda x.x$.

3 The Register-Closure Abstract Machine

So far, we have a "higher-order abstract assembly language" given by the grammar

$$\begin{aligned} C \ &::= \mathbf{halt}\, r \\ &\mid \mathbf{move}\, r_1(\lambda r_2.C) \\ &\mid \mathbf{add}\, r_1 r_2(\lambda r_3.C) \end{aligned}$$

where r, r_1, etc., range over register names (λ-variables).

The *Register-Closure Abstract Machine (RCAM)* is an abstract machine which uses these λ-terms as its assembly language. The RCAM model is based on the following ideas:

- The machine simulates the reduction of a *closure*, that is, a λ-term along with a substitution for its free variables. The behavior of the machine is derived from the definitions of the combinators in its instructions.
- The target machine registers contain the substitution, or (conversely) the substitution represents the contents of the target machine's registers.

More precisely, define a *closure* to be a pair (written $M \bullet \rho$) consisting of a λ-term M and a substitution ρ mapping λ-variables (including the free variables of M) to constants or closed abstractions. The closure $M \bullet \rho$ is a representation of the term obtained by actually performing the substitution ρ on M [3].

By using closures, we need never actually perform a substitution. The key situation is that of beta-reduction:

$$\begin{aligned} ((\lambda x.M)N) \bullet \rho &\equiv ((\lambda x.M) \bullet \rho)(N \bullet \rho) \\ &\rightarrow (M \bullet \rho_x)[x := (N \bullet \rho)] \\ &\equiv M \bullet (\rho[x := (N \bullet \rho)]) \end{aligned}$$

Here $\equiv$ denotes syntactic equivalence, obtained by working out the definition of substitution, $\rightarrow$ denotes reduction, and ρ_x denotes the substitution ρ with x deleted from its domain.

The machine operates by reducing a closed term represented by a closure $C \bullet \rho$. By using the semantics of the combinators, we can work out the reduction rules for the machine for each possible closure $C \bullet \rho$.

For example, let us consider reducing the closure $(\mathbf{add}\, r_1 r_2(\lambda r_3.C) \bullet \rho)$:

$$\begin{array}{rcl}(\mathbf{add}\, r_1 r_2(\lambda r_3.C) \bullet \rho) & \equiv & \mathbf{add}(\rho r_1)(\rho r_2)((\lambda r_3.C) \bullet \rho) \\ & \rightarrow & ((\lambda r_3.C) \bullet \rho)((\rho r_1) + (\rho r_2)) \\ & \rightarrow & C \bullet \rho[r_3 := ((\rho r_1) + (\rho r_2))]\end{array}$$

Thus we can simulate reduction of the underlying term by defining the action of the machine on the **add** instruction as:

$$(\mathbf{add}\, r_1 r_2(\lambda r_3.C) \bullet \rho) \rightarrow C \bullet \rho[r_3 := ((\rho r_1) + (\rho r_2))]$$

Note that the reduction sequence for this closure does just what an addition instruction normally does: it goes on to execute the code sequence C in a substitution where r_3 is bound to the sum of the contents of r_1 and r_2.

The **halt** and **move** instructions similarly turn out to do the expected things. By using the semantics for the **halt** and **move** combinators, we calculate as follows:

$$\begin{array}{rcl}\mathbf{halt}\, r \bullet \rho & \rightarrow & \rho r \\ \mathbf{move}\, r_1(\lambda r_2.C) \bullet \rho & \rightarrow & ((\lambda r_2.C) \bullet \rho)(\rho r_1) \\ & \rightarrow & C \bullet \rho[r_2 := \rho r_1]\end{array}$$

Thus the machine action for these instructions should be:

$$\begin{array}{ll}\mathbf{halt}\, r \bullet \rho & \rightarrow \rho r \\ \mathbf{move}\, r_1(\lambda r_2.C) \bullet \rho & \rightarrow C \bullet \rho[r_2 := \rho r_1]\end{array}$$

The **halt** instruction halts, returning the value in the specified register, and the **move** instruction copies the value in register r_1 to register r_2.

Note that the machine simulates a quasi-leftmost reduction sequence, so the machine is guaranteed to find a normal form of the term on which it starts, if one exists [2].

In general, we can see that the abstract syntax for an instruction that uses registers r_1, r_2, and r_3, and sets register r_4 should be

$$\mathbf{op}\, r_1 r_2 r_3(\lambda r_4.C)$$

and its semantics should be

$$\mathbf{op} = \lambda v_1 v_2 v_3 \kappa . \kappa(f(v_1, v_2, v_3))$$

This easily generalizes to multiple continuations (for branching instructions, or even 0 continuations, as in **halt**), setting multiple registers, or dependence on certain fixed registers. For example, an instruction which branches if the contents of r_1 is non-negative, leaving some information in the fixed register **cc**, might have abstract syntax

$$\mathbf{br}\, r_1(\lambda \mathbf{cc}.C_1)(\lambda \mathbf{cc}.C_2)$$

with semantics

$$\mathbf{br} = \lambda v \kappa_1 \kappa_2 .(v \geq 0) \to \kappa_1(fv), \kappa_2(fv)$$

This easy recipe for specifying both semantics and the format of an instruction makes it easy to incorporate new machine instructions into the RCAM technique. By using continuation semantics, we need not postulate an additional stack mechanism in the machine architecture, beyond what is explicitly manipulated in the instructions.

4 Compiling Addition Expressions

We next turn to the question of writing a compiler with the RCAM machine as its target. Our source language will be addition expressions, given by the grammar

$$\langle \text{exp} \rangle ::= \langle \text{var} \rangle \mid (\langle \text{exp} \rangle + \langle \text{exp} \rangle)$$

We write a continuation semantics that translates this language into λ-terms. As we did for assembly language, we translate variables in the source language into free variables in the λ-terms. The resulting semantics looks very much like Plotkin's algorithm for conversion to continuation-passing style [10]. We write the semantics using $\equiv$ rather than $=$ to emphasize that this is a syntactic transformation:

$$\begin{array}{ll} \mathcal{E}[\![x]\!] & \equiv \lambda \kappa . \kappa x \\ \mathcal{E}[\![(M + N)]\!] & \equiv \lambda \kappa . \mathcal{E}[\![M]\!](\lambda m . \mathcal{E}[\![N]\!](\lambda n . \kappa(m + n))) \end{array}$$

where κ, m, and n are fresh variables.

Our compiler takes two arguments M and K, where M is a source term and K is a λ-term. The output $\mathcal{C}[\![M]\!]K$ will be a λ-term that must meet two requirements:

1. $\mathcal{C}[\![M]\!]K = \mathcal{E}[\![M]\!]K$, where equality means equality under the rules of the λ-calculus, and

2. if C is target code for our machine, then $\mathcal{C}[\![M]\!](\lambda v.C)$ is also code.

Note that $\mathcal{C}[\![-]\!]-$ is a binary (mixfix) operator, and not an application; we have chosen the notation, however, to suggest property (1).

Now we can write the compiler:

$$\begin{array}{ll} \mathcal{C}[\![x]\!]K & \equiv \mathbf{move}\, xK \\ \mathcal{C}[\![(M+N)]\!] & \equiv \mathcal{C}[\![M]\!](\lambda m.\mathcal{C}[\![N]\!](\lambda n.\,\mathbf{add}\, mnK)) \end{array}$$

where all the indicated variables are fresh.

Theorem. $\mathcal{C}[\![-]\!]-$ satisfies its specification.

Proof: Easy structural induction.

To compile an entire program M, we emit $\mathcal{C}[\![M]\!](\lambda v.\,\mathbf{halt}\, v)$. By property (1), this is equal to $\mathcal{E}[\![M]\!](\lambda x.x)$, the cps-transform of M with the identity continuation.

Thus the program $(x+y)+z$ is compiled as follows:

$$\begin{array}{l} \mathcal{C}[\![(x+y)+z]\!](\lambda v.\,\mathbf{halt}\, v) \\ \equiv \mathcal{C}[\![(x+y)]\!](\lambda u.\mathcal{C}[\![z]\!](\lambda w.\,\mathbf{add}\, uw(\lambda v.\,\mathbf{halt}\, v))) \\ \equiv \mathcal{C}[\![(x+y)]\!](\lambda u.\,\mathbf{move}\, z(\lambda w.\,\mathbf{add}\, uw(\lambda v.\,\mathbf{halt}\, v))) \\ \equiv \mathcal{C}[\![x]\!](\lambda a.\mathcal{C}[\![y]\!](\lambda b.\,\mathbf{add}\, ab(\lambda u.\,\mathbf{move}\, z(\lambda w.\,\mathbf{add}\, uw(\lambda v.\,\mathbf{halt}\, v))))) \\ \equiv \mathbf{move}\, x(\lambda a.\,\mathbf{move}\, y(\lambda b.\,\mathbf{add}\, ab(\lambda u.\,\mathbf{move}\, z(\lambda w.\,\mathbf{add}\, uw(\lambda v.\,\mathbf{halt}\, v))))) \end{array}$$

We can see the correspondence to concrete syntax by writing this term as follows:

$$\begin{array}{l} \mathbf{move}\, x\lambda a. \\ \mathbf{move}\, y\lambda b. \\ \mathbf{add}\, ab\lambda u. \\ \mathbf{move}\, z\lambda w. \\ \mathbf{add}\, uw\lambda v. \\ \mathbf{halt}\, v \end{array}$$

This corresponds to the concrete syntax

```
move x, a
move y, b
```

```
add  a, b, u
move z, w
add  u, w, v
halt v
```

Note that, unlike the situation in [15,16,17], there was no need to use associative rules or to rotate trees; the code was generated directly.

Our specifications for a correct compiler enable the compiler writer to use whatever cleverness he or she wishes, so long as the resulting abstract syntax is interconvertible with the code above. For example, by reducing the **move** instructions, we can generate

$$\mathbf{add}\, xy(\lambda u.\, \mathbf{add}\, uz(\lambda v.\, \mathbf{halt}\, v))$$

corresponding to

```
add x, y, u
add u, z, v
halt v
```

Our goal is not to dictate a compiler strategy (*e.g.*, whether this optimized code is generated immediately or by transformations), but rather to give a method for proving the correctness of the chosen strategy. We now consider some strategies for a particular problem: the representation of procedures.

5 Compiling Procedures

In this section we sketch how procedures may be compiled in the RCAM model. The particular compilation scheme is rather simple; at the end of the section we discuss how it may be made more realistic.

We add procedures to the source language by adding the productions

$$\langle \text{exp} \rangle ::= (\lambda \langle \text{var} \rangle . \langle \text{exp} \rangle) \mid (\langle \text{exp} \rangle\ \langle \text{exp} \rangle)$$

with the semantics

$$\begin{aligned} \mathcal{E}[\![\lambda x.M]\!] &\equiv \lambda\kappa.\kappa(\lambda x.\mathcal{E}[\![M]\!]) \\ \mathcal{E}[\![MN]\!] &\equiv \lambda\kappa.\mathcal{E}[\![M]\!](\lambda f.\mathcal{E}[\![N]\!](\lambda a.fa\kappa)) \end{aligned}$$

where κ, f, and a are fresh variables.

To compile these constructs, we must introduce suitable combinators so that the right-hand sides match the general form of the compiler, in which every application is performed by a combinator, and in which every recursive call to the compiler (or semantics) has an abstraction as its second argument. We can do this by introducing the combinators

$$\begin{aligned} \mathbf{close} &= \lambda x\kappa.\kappa x \\ \mathbf{apply} &= \lambda f a\kappa.f a\kappa \\ \mathbf{return} &= \lambda r_1 r_2.r_1 r_2 \end{aligned}$$

With these combinators, we can compile these as follows:

$$\begin{aligned} \mathcal{C}[\![(\lambda x.M)]\!]K &\equiv \mathbf{close}(\lambda x\kappa.\mathcal{C}[\![M]\!](\lambda v.\,\mathbf{return}\,\kappa v))K \\ \mathcal{C}[\![(MN)]\!]K &\equiv \mathcal{C}[\![M]\!](\lambda f.\mathcal{C}[\![N]\!](\lambda a.\,\mathbf{apply}\,faK)) \end{aligned}$$

where all the indicated variables are fresh. It is now easy to confirm, by structural induction, that this compiler still satisfies its specification.

The formats for our three additional instructions are

$$\begin{aligned} C \; ::= \; & \mathbf{close}(\lambda x\kappa.C)(\lambda r.C') \\ \mid \; & \mathbf{apply}\, r_1 r_2(\lambda r_3.C) \\ \mid \; & \mathbf{return}\, r_1 r_2 \end{aligned}$$

By examining the definitions of the combinators, we can now work out the behavior of the machine on these instructions. The **close** instruction creates a closure of two arguments which represents a procedure and stores the closure in the target register.

$$\begin{aligned} \mathbf{close}(\lambda x\kappa.C)(\lambda r.C') \bullet \rho &\rightarrow ((\lambda r.C') \bullet \rho)((\lambda x\kappa.C) \bullet \rho) \\ &\rightarrow C' \bullet \rho[r := ((\lambda x\kappa.C) \bullet \rho)] \end{aligned}$$

Note that the **close** instruction is just another format for **move**; we give it a different name because it behaves somewhat differently: it saves the current register set.

The **apply** instruction invokes such a procedure object. It creates a closure to serve as the continuation, and applies the procedure object to the argument and the continuation. In the following, assume that $\rho r_1 = (\lambda x\kappa.C') \bullet \rho'$. Then we reduce as follows:

$$\begin{aligned}\textbf{apply}\, r_1 r_2 (\lambda r_3.C) \bullet \rho &\rightarrow (\rho r_1)(\rho r_2)((\lambda r_3.C) \bullet \rho)\\ &\rightarrow ((\lambda x\kappa.C') \bullet \rho')(\rho r_2)((\lambda r_3.C) \bullet \rho)\\ &\rightarrow C' \bullet \rho'[x := \rho r_2; \kappa := ((\lambda r_3.C) \bullet \rho)]\end{aligned}$$

Thus the body of the procedure is executed, with the actual parameter placed in the register that the procedure expects, and the continuation (with the caller's registers saved) placed in the callee's continuation register.

The **return** instruction returns from such a procedure. In the following, assume that in ρ, r_1 contains the continuation $(\lambda r_3.C') \bullet \rho'$.

$$\begin{aligned}\textbf{return}\, r_1 r_2 \bullet \rho &\rightarrow (\rho r_1)(\rho r_2)\\ &\rightarrow ((\lambda r_3.C') \bullet \rho')(\rho r_2)\\ &\rightarrow C' \bullet \rho'[r_3 := \rho r_2]\end{aligned}$$

Thus the caller's registers are restored, and the result placed in r_3.

This procedure mechanism relies on microcode to store the register set (which may or may not be realistic). Our techniques also extend to modelling the representation of procedures by explicit closures, created by the compiler designer, but that is beyond the scope of this paper.

We can summarize the behavior of our machine as follows:

$$\begin{array}{ll}\textbf{move}\, r_1(\lambda r_2.C) \bullet \rho & \rightarrow C \bullet \rho[r_2 := \rho r_1]\\ \textbf{add}\, r_1 r_2(\lambda r_3.C) \bullet \rho & \rightarrow C \bullet \rho[r_3 := ((\rho r_1) + (\rho r_2))]\\ \textbf{halt}\, r \bullet \rho & \rightarrow \rho r\\ \textbf{close}(\lambda x\kappa.C)(\lambda r.C') \bullet \rho & \rightarrow C' \bullet \rho[r := ((\lambda x\kappa.C) \bullet \rho)]\\ \textbf{apply}\, r_1 r_2(\lambda r_3.C) \bullet \rho & \rightarrow C' \bullet \rho'[x := \rho r_2; \kappa := ((\lambda r_3.C) \bullet \rho)]\\ & \quad (\rho r_1 = (\lambda x\kappa.C') \bullet \rho')\\ \textbf{return}\, r_1 r_2 \bullet \rho & \rightarrow C' \bullet \rho'[r_3 := \rho r_2]\\ & \quad (r_1 = (\lambda r_3.C') \bullet \rho')\end{array}$$

We can no longer argue that the machine simulates a quasi-leftmost reduction, since the procedure call and return instructions work correctly only if the term has the proper form. We can, however, prove completeness relative to an operational semantics for the source language, in the style of [10]: We can give a call-by-value operational semantics (similar to eval_V in [10]), and prove a simulation theorem (similar to [10]'s Theorem 6.2); the proof is simpler because the compiler does all the "administrative reductions." The details will be included in the final version of the paper.

6 Comparison with Previous Work

This work is an elaboration of our work in the early 1980's on combinator-based compiler generation [15,16,17]. The RCAM machine improves on this work by replacing combinator terms with more general continuation-passing terms using variables, thus obtaining a more direct connection with register architectures.

The idea of using the conversion to continuation-passing style as the basis for compilation is an old one [13,14,6, among others]. The most prominent recent work in this area is that of [1,5]. Both of these papers use cps code as an abstract assembly language, though neither use continuation semantics as a way of generating the cps code ([1] uses a direct semantics to generate terms which are then passed through a cps conversion algorithm; [5] transforms source expressions into cps terms but then translates into a target language whose syntax is a subset of cps "but with a completely different semantics" [5, p. 281]). The RCAM machine may be regarded as providing a formal basis for the connection between cps code and machine code. Because one has a formal basis, it much easier to adapt the approach to different languages and architectures.

Our use of semantics is unusual in its use of concrete semantics. This is in contrast to the usual view that semantics maps to some domain of meanings. The idea that semantics should have an explicit syntactic element is found in [12] and in Pleban and Lee's macrosemantics [8]. In our case, it was not necessary to specify a domain of meanings for the output terms of the semantics; any λ-model would suffice. If our compiler needed to reason about addition, then we could re-introduce traditional semantics through the technique of altering our compiler correctness criterion to replace the requirement of interconvertibility by equality in some specific model or class of models.

A second unusual feature of our semantics is its use of free variables in place of the more conventional environments. This style of translation goes back as least as far as [11], though it is implicit in the principle of Church-style encoding. It is possible to adapt the RCAM model to use a single environment register (as is used in, say [15]); on the other hand, our approach allows more explicit treatment of register allocation and environment representation.

This paper presents the key ideas of the RCAM method. We are currently in the process of applying these techniques to a variety of situations, including explicit procedure representations, explicit environment representations, imperative code, loops, and recursions. We hope to report on these later.

References

[1] Appel, A.W., and Jim, T., "Continuation-Passing, Closure-Passing Style," *Conf. Rec. 16th ACM Symp. on Principles of Programming Languages* (1989), 293–302.

[2] Barendregt, H.P. *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, Amsterdam, 1981.

[3] Boyer, R.S., and Moore, J S. "The Sharing of Structure in Theorem-Proving Programs," in *Machine Intelligence 7* (B. Meltzer & D. Michie, eds), Edinburgh University Press (1972), 101–116.

[4] Church, A. "A formulation of the simple theory of types," J. of Symbolic Logic 5 (1940), 56–68.

[5] Kelsey, R., and Hudak, P. "Realistic Compilation by Program Transformation," *Conf. Rec. 16th Ann. ACM Symp. on Principles of Programming Languages* (1989), 281–292.

[6] Kranz, D.A., Kelsey, R., Rees, J.A., Hudak, P., Philbin, J., and Adams, N.I., "Orbit: An Optimizing Compiler for Scheme," *Proc. SIGPLAN '86 Symp. on Compiler Construction*, *SIGPLAN Notices 21*(7), July, 1986, 219-223.

[7] McCarthy, J. "Towards a Mathematical Science of Computation," *Information Processing 62* (Popplewell, ed.) Amsterdam:North Holland, 1962, 21–28.

[8] Lee, P., and Pleban, U., "On the Use of LISP in Implementing Denotational Semantics," *Proc. 1986 ACM Conf. on Lisp and Functional Programming*, 233–248.

[9] Pfenning, F., and Elliott, C., "Higher-Order Abstract Syntax," *Proc. SIGPLAN '88 Conf. on Prog. Lang. Design and Implementation*, (June, 1988), 199–208.

[10] Plotkin, G.D. "Call-by-Name, Call-by-Value and the λ-Calculus," *Theoret. Comp. Sci. 1* (1975) 125–159.

[11] Reynolds, J.C. "The Essence of Algol," in *Algorithmic Languages*, (J. W. deBakker and J.C. van Vliet, eds.) North-Holland, Amsterdam, 1981, pp. 345–372.

[12] Sethi, R. "Control Flow Aspects of Semantics-Directed Compiling" *ACM Trans. on Prog. Lang. and Sys. 5* (1983) 554–596.

[13] Steele, G.L. "Rabbit: A Compiler for Scheme," MIT AI Memo No. 474 (May, 1978).

[14] Wand, M. and Friedman, D.P. "Compiling Lambda Expressions Using Continuations and Factorizations," *J. of Computer Languages 3* (1978), 241–263.

[15] Wand, M. "Semantics-Directed Machine Architecture" *Conf. Rec. 9th ACM Symp. on Principles of Prog. Lang.* (1982), 234–241.

[16] Wand, M. "Deriving Target Code as a Representation of Continuation Semantics" *ACM Trans. on Prog. Lang. and Systems 4*, 3 (July, 1982) 496–517.

[17] Wand, M. "Loops in Combinator-Based Compilers," *Conf. Rec. 10th ACM Symposium on Principles of Programming Languages* (1983), 190–196.