Generating Efficient Code from Continuation Semantics*

Mikael Pettersson

Department of Computer and Information Science, Linköping University

S-581 83 Linköping, Sweden

mpe@ida.liu.se

ABSTRACT

In this paper we present a method for generating very efficient code from continuation-style denotational semantics. We claim that this form arises naturally in the definition of procedural languages, such as C. Once the language semantics has been simplified using a compile-time versus run-time distinction, we recognize two important properties of the resulting form, *single-threading* and *label-freeness of closures*. The first allows a conventional call-by-value execution with a single store. The second property is central to our work, as it allows bindings and closure creation to be replaced by assignments to uniquely named variables, i.e. quadruples. These quadruples are then passed on to a conventional code optimizer and generator. As far as we know, this is the first time a quality code compiler has been generated which does not rely on either β-reductions or an abstract stack machine.

1. Introduction

Given a standard denotational definition of a programming language, consider the task of generating a compiler for it which produces high-quality code. The usual approach for deriving a prototype compiler results in code being generated for a stack machine [Schmidt85,Paulson82]. However, this has two drawbacks. First one needs to actually derive a stack machine from the denotational equations, which may require some non-trivial amount of work [Wand82]. Second, the generated code is not in a form suitable for modern register-oriented machines.

Why then is the stack machine so popular? To see this, we will consider the case of generating code for an addition expression:

$$R[\![E_1+E_2]\!]\rho\kappa \quad = \quad R[\![E_1]\!]\rho\{\lambda\varepsilon_1.R[\![E_2]\!]\rho\{\lambda\varepsilon_2.\kappa(\varepsilon_1+\varepsilon_2)\}\}$$

(Let us assume that the environment ρ and continuation κ are compile-time constants.) The central issue is how to communicate the value 'ε_1' past the evaluation of the second expression to the addition operator, without using expensive run-time closures. A simple solution is to use a stack

* This work was supported by the Swedish Board for Technical Development (STU).

to send (sequences of) values around. Producing a value corresponds to a "push", and consuming a value is done by a "pop".

Our solution however is based on the following observation: whenever a value is produced, we know that there is a receiving continuation $\kappa=\lambda x.0$. Moreover, the text of this continuation is usually *known at compile-time*. So, the application of this continuation to a value ε β-reduces to $0[\varepsilon/x]$. This is directly analogous to the imperative code $x:=\varepsilon$; 0. Applying this idea throughout results in a quadruple-style code generator.

The code generation algorithm can then be formulated as these steps:

1. Calculate the meaning of the program, yielding a continuation-style λ-term
2. α-convert this term so that all bindings have unique names (this allows us to dispense with run-time closures, as we show later)
3. Traverse the converted term, and produce quadruples as shown above.

The quadruples can then be fed into a conventional optimizing back-end for global optimization and register allocation [ASU86,Wall87].

Overview of the paper

Our work is based on an interpretive semantics for a subset of C, which is given in Appendix A. The subset involves integer-only variables, recursive functions of one argument, assignment and conditional statements, loops and nested declarations. The issues of input/output are ignored in this paper. Our notation for describing the semantics is standard [Stoy77].

We first apply some simple transformations, separation of static and dynamic semantics and separation of compile-time and run-time entities, to arrive at a simplified interpretive semantics. We then choose a trivial representation of the interpretive semantics, isomorphic to the text of the original λ-terms. An important property of this representation is identified, *label-freeness of closures*, and it is shown that run-time closures can be eliminated. A simple machine semantics is given, followed by an algorithm for converting the intermediate λ-representation to quadruples. Finally, we show a code generation example, and relate this approach to other work.

2. Simplification

The dynamic semantics is in general quite cluttered with random details concerning three conceptually disjoint events: type checking, meaning calculation and execution. The goal of this simplification step is to split these events into separate modules or passes.

Static Analysis

Type checking is done to check that identifiers are properly declared, that expressions have the correct type depending upon their use and so on. A type error discovered by the semantics causes *Wrong* to be applied and, conceptually, the execution to be terminated. However, as with all other languages that associate types with declarations and not with values only, most type-checking can be done at compile-time.

The static semantics for TINY-C can be derived in the following way. A *Type* environment maps identifiers to a description of the type the value of the identifier would have at run-time. The

valuation functions, following the same structure as the original semantics, use this environment to assign type information to all the program clauses. Finally, the top-level valuation returns true (false) if the program is seemingly type-correct (-incorrect). The interpretive semantics can be factored into a static and a dynamic part, thus simplifying the definition.

Semantic Domains:

$t \in Type = \{t\_int, t\_func, t\_loc, t\_kont, t\_err, t\_ok\}^\circ$
$\upsilon \in TEnv = Ide \to Type$

Semantic Equations:

RS : $Exp \to TEnv \to Type$
PS : $Cmd \to TEnv \to Type$
DS : $Decl \to TEnv \to TEnv \times Type$
MS : $Prog \to Bool$ [*returns true iff the program is type-safe*]

$RS[\![N]\!]\upsilon = t\_int$
$RS[\![I]\!]\upsilon = \upsilon[\![I]\!]=t\_loc \to t\_int \,[\!]\, t\_err$
$RS[\![I(E)]\!]\upsilon = \upsilon[\![I]\!]=t\_func \wedge RS[\![E]\!]\upsilon=t\_int \to t\_int \,[\!]\, t\_err$
$RS[\![E_1 \Omega E_2]\!]\upsilon = RS[\![E_1]\!]\upsilon=t\_int \wedge RS[\![E_2]\!]\upsilon=t\_int \to t\_int \,[\!]\, t\_err$

$PS[\![I:=E]\!]\upsilon = \upsilon[\![I]\!]=t\_loc \wedge RS[\![E]\!]\upsilon=t\_int \to t\_ok \,[\!]\, t\_err$
$PS[\![\text{if } E\ \Gamma_1\ \Gamma_2]\!]\upsilon = RS[\![E]\!]\upsilon=t\_int \wedge PS[\![\Gamma_1]\!]\upsilon=t\_ok \to PS[\![\Gamma_2]\!]\upsilon \,[\!]\, t\_err$
$PS[\![\text{while } E\ \Gamma]\!]\upsilon = RS[\![E]\!]\upsilon=t\_int \to PS[\![\Gamma]\!]\upsilon \,[\!]\, t\_err$
$PS[\![skip]\!]\upsilon = t\_ok$
$PS[\![\text{return } E]\!]\upsilon = \upsilon[\![\text{return}]\!]=t\_kont \wedge RS[\![E]\!]\upsilon=t\_int \to t\_ok \,[\!]\, t\_err$
$PS[\![\Gamma_1;\Gamma_2]\!]\upsilon = PS[\![\Gamma_1]\!]\upsilon=t\_ok \to PS[\![\Gamma_2]\!]\upsilon \,[\!]\, t\_err$
$PS[\![\Delta;\Gamma]\!]\upsilon = let\ \langle\upsilon',\iota\rangle=DS[\![\Delta]\!]\upsilon\ in\ \iota=t\_ok \to PS[\![\Gamma]\!]\upsilon' \,[\!]\, t\_err$

$DS[\![\text{int } I_1(\text{int } I_2)\Gamma]\!]\upsilon = let\ \upsilon'=\upsilon[t\_func/[\![I_1]\!]]\ in\ \langle\upsilon', PS[\![\Gamma]\!]\upsilon'[t\_loc/[\![I_2]\!]]\rangle$
$DS[\![\text{int } I]\!]\upsilon = \langle\upsilon[t\_loc/[\![I]\!]], t\_ok\rangle$
$DS[\![\Delta_1\ \Delta_2]\!]\upsilon = let\ \langle\upsilon',\iota\rangle=DS[\![\Delta_1]\!]\upsilon\ in\ \iota=t\_ok \to DS[\![\Delta_2]\!]\upsilon' \,[\!]\, \langle\upsilon',\iota\rangle$

$MS[\![\Delta;\Gamma]\!] = let \langle\upsilon,\iota\rangle=DS[\![\Delta]\!](Prelude_\upsilon)\ in\ \iota=t\_ok \to PS[\![\Gamma]\!]\upsilon=t\_ok \,[\!]\, false$

Auxiliary Functions:

$Prelude_\upsilon : TEnv$ [*implementation dependent*]

Table 1. Static Semantics

Compile-time entities

A compile-time entity is one that needs no access to the state or any input data for its value to be determined. In a conventional compiler, the environment would correspond to the symbol table and be a compile-time object. Unfortunately, the environment carries two run-time types: the locations of declared variables, and the return continuations to functions.

Looking first at the return κ's to function calls, we may notice their restricted use. On function entry, the κ is bound in the environment so that `return` may find it. Thus all function bodies have the return point as a free variable. With a little operational intuition, we can see that a stack of return points can be added to the state, together with two simple primitives *Call* and *Return* to maintain it.

This leaves us with the environment containing run-time locations. In general, the compiler has to resort to being very intimate with the memory allocation mechanisms of the target machine, introducing notions such as *stack offsets* or *frame pointers*. While manageable, this is not an attractive solution due to the amount of details involved. We have chosen not to follow this path, as it turns out that our handling of temporary bindings equally well takes care of the variable locations. Also, we do *not* have to assume a certain stack model of memory allocation.

Simplified Interpretive Semantics

The simplified definition consists of the original definition, where the static semantics have been removed (i.e., no error checks), and the modification described above regarding the stack of return continuations. We have, to simplify the next step, put the equations in a kind of "normal" form (not to be confused with normal form of λ-terms). First, all primitive operators (like *Fetch*, *Cond* etc.) are uncurried, so all expression continuations are explicitly written as $(\lambda\varepsilon.\theta)$. Also, for those cases where expression continuations are applied directly, we have introduced an explicit operator $Literal = \lambda\varepsilon.\lambda\kappa.\kappa\varepsilon$.

Semantic Domains:

$\phi \in Func = Kont$
$\delta \in DVal = Loc + Func + Void$
$\sigma \in State = (Loc \to EVal) \times Kont^*$

Domain Operations:

$empty_\sigma = \langle(\lambda\alpha.Unspecified),\langle\rangle\rangle$
$Fetch\ \alpha\kappa = \lambda\langle map,\kappa^*\rangle.\kappa(map\ \alpha)\langle map,\kappa^*\rangle$
$Update\ \alpha\varepsilon\theta = \lambda\langle map,\kappa^*\rangle.\theta\langle map\ [\varepsilon/\alpha],\kappa^*\rangle$
$Call : Func \to EVal \to Kont \to Cont$
$Call\ \phi\varepsilon\kappa = \lambda\langle map,\kappa^*\rangle.\phi\varepsilon\langle map,\kappa::\kappa^*\rangle$
$Return : Kont$
$Return\ \varepsilon = \lambda\langle map,\kappa::\kappa^*\rangle.\kappa\varepsilon\langle map,\kappa^*\rangle$
$Literal : EVal \to Kont \to Cont$
$Literal\ \varepsilon\kappa = \kappa\varepsilon$

Semantic Equations:

$R[\![N]\!]\rho\kappa = Literal\ [\![N]\!]\kappa$
$R[\![I]\!]\rho\kappa = Fetch\ ((\rho[\![I]\!])|Loc)\kappa$
$R[\![I(E)]\!]\rho\kappa = R[\![E]\!]\rho\{\lambda\varepsilon.Call\ ((\rho[\![I]\!])|Func)\varepsilon\kappa\}$

$P[\![I:=E]\!]\rho\theta = R[\![E]\!]\rho\{\lambda\varepsilon.Update\ ((\rho[\![I]\!])|Loc)\varepsilon\theta\}$
$P[\![return\ E]\!]\rho\theta = R[\![E]\!]\rho\{\lambda\varepsilon.Return\ \varepsilon\}$

Auxiliary functions:

$UserFunc\ [\![I]\!][\![\Gamma]\!]\rho = \lambda\varepsilon_1.New\ \{\lambda\varepsilon_2.Update\ \varepsilon_2\varepsilon_1\{P[\![\Gamma]\!]\rho'\{Literal\ (Unspecified)\{\lambda\varepsilon_3.Return\ \varepsilon_3\}\}\}\}$
$\quad where\ \rho' = \rho[((\varepsilon_2\ as\ Loc)\ in\ DVal)/[\![I]\!]]$

Table 2. Revised Dynamic Semantics
(*only the differences are listed here*)

3. Viewing λ as Assign-then-Goto

Now that we have a simplified interpretive semantics, it is time to recall our original intuition: to recognize that $(\lambda x.0)\varepsilon$ β-reduces to $0[\varepsilon/x]$, and to transform this to $x{:=}\varepsilon;\ 0$. A key element here is that we want to dispense with run-time closures. We have to be careful though so that we do not introduce name conflicts. Given the definitions so far, the denotation for 1+(2+3) is:

$$\lambda\rho.\lambda\kappa.\text{Literal}[\![1]\!](\lambda\varepsilon_1.(\lambda\kappa.\text{Literal}[\![2]\!](\lambda\varepsilon_1.\text{Literal}[\![3]\!](\lambda\varepsilon_2.\kappa(\varepsilon_1{+}\varepsilon_2))))(\lambda\varepsilon_2.\kappa(\varepsilon_1{+}\varepsilon_2)))$$

which, given an environment and a continuation λx.0, we would transform into:

$$\varepsilon_1 := 1;\ \varepsilon_1 := 2;\ \varepsilon_2 := 3;\ \varepsilon_2 := \varepsilon_1 + \varepsilon_2;\ x := \varepsilon_1 + \varepsilon_2;\ 0$$

This is incorrect, since variables are overwritten over and over again. If we assume that all λ-terms are α-converted to have unique bound names, then this problem is eliminated.

Proof Outline

Here we will show that run-time closures can indeed be dispensed with if all bound temporaries have unique names. Our informal proof is done in two steps. First we show the result for programs without user-defined function, i.e. the program is just a command sequence. Then we discuss why functions complicate matters, and a simple and effective solution which handles that case as well.

Part 1: No Functions

It is convenient to have these definitions handy:

- an *open* term is a λ-term of type κ or 0 with free variables, otherwise it is *closed*
- a term is *labeled* if there is more than one reference to it, otherwise it is *label-free*
- if the term B is a continuation to a term A, then A is a *predecessor* of B
- the use of a name is *sound* if it is preceded by the corresponding definition (binding, assignment), otherwise, it is *unsound*

Our correctness criterion can then be stated as:

All uses of variables must be sound.

Intuitively, open terms need a run-time environment to interpret their free variables. If we see the control flow as a directed graph, then a node (term) is labeled if there are multiple arcs pointing to it. In linearized code, all but one of these arcs must be expressed as a `goto` to the label.

Consider an open, label-free term x. Since it is label-free, it has at most *one* predecessor y. If y is closed, then it must bind the free variables of x, and thus all uses in x are sound. Otherwise, if y is open, then it can be either labeled or label-free. If it is labeled, then uses in y (and x) can be unsound. If it is label-free, then this argument applies again, and the uses are sound.

The upshot of this reasoning is that the correctness criterion is fulfilled if:

All open terms are label-free

So let us first examine all contexts in the revised semantics where some term might be labeled.

Semantic Equations: The `if` command uses its continuation twice, since it is the continuation of both the true and false branches. So, command *denotations* (not continuations 0 in general) may

be labeled. The denotation of the `while` command is labeled since it refers back to itself, while at the same time being referred to by its predecessor. The only other bound variables that have multiple uses are the environment arguments ρ, but they are compile-time only objects.

Now that we know that only command denotations can be labeled, we need to know if they have free variables. The answer is both yes and no. No, because each right-hand-side of the semantic equations mentions only bound variables (i.e., all denotations are closed). Yes, because the locations of declared (in TINY-C) variables are bound in the environment; if we dispense with the environment then these locations become free to the commands and expressions in their scope.

However, here we are saved by the structured nature of our language. While commands may be labeled, and also have locations as free variables, the restrictions on where jumps may go ensure that uses are sound anyway. Since jumps may only occur *within* the same scope level and never *into* a more deeply nested scope, we see that all uses are indeed preceded by their definitions[1]. A quick look at the equations for `if` and `while` reveals that all commands involved execute in the same scope, so jumps here cannot cause unsound references to declared locations.

Implementability

The natural implementation of temporaries as those in quadruples is to put them in the state — they are global anyway. We can add a third component $Env = Ide \rightarrow EVal$ to keep track of defined temporaries. Since all uses are sound, this is a total function. Also, since all names are unique and the component is part of the single-threaded state, it can be efficiently implemented as an array indexed by the names of the temporaries.

Part 2: Functions

Functions pose a problem because they may call themselves. Consider the following example:

```
int fac(int n) {
    if ( n == 0 )
        return 1;
    else
        return n * fac(n-1);     (*)
}
```

The recursive call on the marked line causes `fac` to be entered again, overwriting the live temporaries. On return, the temporaries in the first invocation of the function will have incorrect values. Apparently, we need infinitely many copies of the function, all with unique names for their temporaries.

A modification to the state again would take care of this problem. The second component, the return points, is extended to also contain the values the temporaries in the called function had on entry. On return, these temporaries are restored, i.e. a callee-saves protocol.

[1] Even "exit" jumps such as `break`, `continue` and `resultis` are sound. The more general formulation is to require that all jumps transfer control to an already active scope. Downwards jumps may be handled by changing the denotation of labels to include code to set up the proper environment, or [as in the original C compiler] by flattening the declaration structure so that all code in a function body execute in the same scope.

4. Target Representation

Now that we have shown the soundness of our approach, it is time to define a concrete representation to be used by the compiling semantics. Since our idea of quadruples was expressed by syntactic transformations of λ-terms, we will represent them by a concrete datatype of Λ-terms, that is isomorphic to the text of the λ-terms in the simplified interpretive semantics.

Expression continuations $\{\lambda\varepsilon.0\}$ are written $(\Lambda I.0)$. Command continuations are written as before, but with references to values ε replaced by identifiers I. It is assumed that the compiling semantics, when it creates new terms with bindings, allocates new unique names for these bindings. Also, the compile-time environment ρ is changed to bind the temporary assigned to hold locations, rather than the locations themselves. The representation of functions now includes a list of its temporaries so that the callee-saves protocol can be used. The compiler specification is given in appendix B, together with an execution semantics in appendix C.

The Abstract Machine

The job of the machine is to transform Λ-terms and states in a manner faithful to the β-reductions of λ-terms. As an example, let us show how the *Fetch* primitive is handled, by observing its defined behavior:

$$Fetch\ \alpha\{\lambda\varepsilon.0\}\langle map, \kappa^*\rangle \rightarrow_\beta 0[(map\ \alpha)/\varepsilon]\langle map, \kappa^*\rangle$$

Thus the machine action is:

$$\mathrm{Exec}[\![Fetch\ I_1\ (\Lambda I_2.0)]\!]\langle\sigma, r^*, \rho\rangle \quad = \quad \mathrm{Exec}[\![0]\!]\langle\sigma, r^*, \rho[(\sigma(\rho I_1))/I_2]\rangle$$

The other primitives are treated similarly, as shown in the appendix.

Since this machine simulates a head reduction sequence, it follows that the machine finds a head normal form for the program iff there is one [Barendregt84].

The Code Generation Algorithm

Given a (possibly) circular Λ-term, the algorithm below translates it to sequences of assignments, simple arithmetic, tests and jumps. We formulate the algorithm for code generation of a command denotation, since details about how functions and temporaries are declared in the quadruple language are ignored here.

The input term is assumed to meet the following requirements:

- each node is "unmarked"
- each node has a reference count field containing the number of references to the node
- the "false" branch of *Cond* nodes is assumed to have a reference count greater than one (this is just to force a label definition for the node, and simplify code generation)

Quadruples are written as [*text*]; they should be self-explanatory.

```
Gen(node) {
  if the node is marked, then
    output [goto L], where L is the label of the node, and return
  endif
  mark the node
  if the node's reference count > 1, then
    allocate a new label L, and store it in the node
    output [label L:]
  endif
  dispatch on the structure of the node:
    [Halt] => output [halt].
    [New (ΛI.θ)] => output [I := new], then Gen(θ).
    [Cond I θ1 θ2] => output [if not I then goto L], where L is the label
                      of θ2, call Gen(θ1), then Gen(θ2).
    [Literal v (ΛI.θ)] => output [I := v], then Gen(θ).
    [Call φ I1 (ΛI2.θ)] => output [I2 := φ(I1)], then Gen(θ).
    [Return I] => output [return I].
    [Binary + I1 I2 (ΛI3.θ)] => output [I3 := I1 + I2], then Gen(θ).
    (similarly for the other binary operators)
    [Update I1 I2 θ] => output [mem[I1] := I2], then Gen(θ).
    [Fetch I1 (ΛI2.θ)] => output [I2 := mem[I1]], then Gen(θ).
  end dispatch
}
```

5. Related Work

As mentioned in the introduction, we consider our approach of generating quadruples from continuation-style semantics to be quite different from the usual stack-based systems.

Wand [Wand82] proposes that closures be eliminated by introducing a family D_k of "argument-steering" combinators, similar to $B=\lambda f.\lambda g.\lambda x.f(gx)$. The result is a tree labeled with D:s, which can be rotated to a nice linear form. Then a stack-machine is derived, using the same kind of analysis as done here. Performance results were not given, but since his system does not do β-reductions at run-time, it should be fairly good. A disadvantage is that the resulting compiling semantics looks rather different from the original one.

Sethi [Sethi81] arrives at essentially the same kind of machine as Wand, but uses a more intuitive notion of "pipes" of values rather than specialized combinators.

Appel's work [Appel85] is in some sense similar to ours. He too generates register-transfers (or quadruples) instead of stack code. However, his system is built using very intimate knowledge about compile-time versus run-time entities, and many highly specific reduction rules. The code generated is similar in quality as ours, with many temporaries which must be optimized away by a code optimizer.

Most other systems appear to either be very general, relying on essentially interpreting λ-calculus (eg. [Paulson82]), or very specialized, using lots of detailed knowledge about "standard semantics" (eg. [Raskovsky82]).

6. Implementation Status

We started with a semantics-directed interpreter for our small C subset, written in C++ [Pettersson89]. The modifications to the interpretive definition described in this paper were then applied, resulting in a semantics-directed compiler. The quadruples are written as statements in C, and then compiled by an optimizing C compiler (we use GCC [Stallman89]). The practicality of treating C as an UNCOL is supported by the fact that languages such as C++, Eiffel, Modula-3, BASIC, Pascal, Fortran, Lisp and Scheme all have implementations taking this route.

A Code Generation Example

To illustrate the effectiveness of our approach, we will consider the recursive factorial function shown earlier. Our code generator emits the following quadruples:

```
int fac(int temp1) {
    int temp2, temp3, temp4, temp5, temp6, temp7, temp8;
    int temp9, temp10, temp11;
    temp2 = temp1;
    temp3 = 0;
    temp4 = temp2 == temp3;
    if (! temp4) goto L1;
    temp5 = 1;
    return temp5;
L1: temp6 = temp1;
    temp7 = temp1;
    temp8 = 1;
    temp9 = temp7 - temp8;
    temp10 = fac(temp9);
    temp11 = temp6 * temp10;
    return temp11;
}
```

The code is then compiled by GCC on a Sun-3 yielding this assembly output:

```
_fac: movel d2,sp@-
      movel sp@(8),d2
      seq d0
      btst #0,d0
      jeq L3
      moveq #1,d0
      jra L1
L3:   movel d2,d1
      subql #1,d1
      movel d1,sp@-
      jbsr _fac
      mulsl d2,d0
      addqw #4,sp
L1:   movel sp@+,d2
      rts
```

7. Experiences

Implementing the code generator was very easy, as might be expected. The final code quality is close to what an optimizing compiler would have generated. Due to the restricted nature of our language, large applications have not been compiled or benchmarked. But given that quadruples are well-known technique in the compiler community, we see no reason to expect any problems in extending this work for full Algol-like languages.

8. Conclusions

A semantics-directed compiler has been derived for a small procedural language using a continuation-style denotational semantics. The key result is the definition of a direct mapping from continuation-style λ-terms to quadruples. The mapping is shown to be sound. Using a standard optimizing code generator as the back-end, high quality code generation can be achieved.

As far as we know, this is the first time a semantics-directed compiler has been derived which does not rely on β-reductions or an idealized stack machine. Compared to the stack machine approach, ours is more direct and slightly less operational. It is also closer to the characteristics of modern hardware, which emphasizes the use of registers rather than a stack.

9. References

[Appel85] Appel, Andrew W. *Compile-time Evaluation and Code Generation for Semantics-directed Compilers* (Ph.D. thesis, Carnegie-Mellon Univ., 1985)

[ASU86] Aho, A. V., Sethi, R. and Ullman J. D. *Compilers Principles, Techniques and Tools* (Addison-Wesley, 1986)

[Barendregt84] Barendregt, H.P. *The Lambda Calculus, Its Syntax and Semantics* (Revised Edition, North-Holland, 1984)

[Paulson82] Paulson, L. *A Semantics-Directed Compiler Generator* (Proceedings 9th ACM Conference on Principles of Programming Languages, p. 224-233)

[Pettersson89] Pettersson, M. *Generating Interpreters from Denotational Definitions using C++ as a Meta-language* (Report LiTH-IDA-R-89-52, Linköping Univ, 1989)

[Raskovsky82] Raskovsky, Martin R. *Denotational Semantics as a Specification of Code Generators* (Proceedings ACM SIGPLAN 82 Conference on Compiler Construction, p. 230-244)

[Schmidt85] Schmidt, David A. *An implementation from a direct semantics definition*, in: N.D. Jones, Ed., Proc. Workshop on Programs as Data Objects (LNCS 217, Springer, Berlin, 1985).

[Sethi81] Sethi, Ravi. *Control flow aspects of semantics directed compiling* (CSTR-98, Bell Labs, 1981)

[Stallman89] Stallman, Richard M. *Using and Porting GNU CC* (Free Software Foundation, 1989)

[Stoy77] Stoy, Joseph E. *Denotational Semantics* (MIT Press, Cambridge, MA, 1977)

[Wall87] Wall, David A. and Powell, Michael L. *The Mahler Experience: Using an Intermediate Language as the Machine Description* (DECWRL Technical Report 87/1, 1987)

[Wand82] Wand, Mitchell. *Semantics-Directed Machine Architecture* (Proceedings 1982 ACM Conference on Lisp and Functional Programming)

Appendix A: Interpretive Semantics for TINY-C

Syntactic Domains:

I	$\in$	Ide	(identifiers)
N	$\in$	Nml	(numerals)
E	$\in$	Exp	(expressions)
Ω	$\in$	Ops	(binary operators)
Γ	$\in$	Cmd	(commands)
Δ	$\in$	Decl	(declarations)
Ψ	$\in$	Prog	(programs)

Abstract Syntax:

Ψ	::=	$\Delta;\Gamma$
Δ	::=	int I_1(int I_2) Γ | int I | Δ_1 Δ_2
Γ	::=	I := E | if E Γ_1 Γ_2 | while E Γ | skip | $\Gamma_1;\Gamma_2$ | $\Delta;\Gamma$ | return E
E	::=	E_1 Ω E_2 | I(E) | I | N
Ω	::=	$+ \mid - \mid \times \mid + \mid = \mid \leq \mid < \mid \neq \mid > \mid \geq$

Semantic Domains:

		Answer	=	$\{ Ok \}^{\circ} + Err$
ν	$\in$	*Int*	=	$\{ \cdots, -2, -1, 0, 1, 2, \cdots \}^{\circ}$
α	$\in$	*Loc*	=	*Int*
ϕ	$\in$	*Func*	=	$Kont \rightarrow Kont$
		Void	=	$\{Unknown\}^{\circ}$
β	$\in$	*SVal*	=	$Int + Void$
δ	$\in$	*DVal*	=	$Loc + Func + Kont + Void$
ε	$\in$	*EVal*	=	*Int*
σ	$\in$	*State*	=	$Loc \rightarrow SVal$
θ	$\in$	*Cont*	=	$State \rightarrow Answer$
κ	$\in$	*Kont*	=	$EVal \rightarrow Cont$
χ	$\in$	*Dcont*	=	$Env \rightarrow Cont$
ρ	$\in$	*Env*	=	$Ide \rightarrow DVal$
X	$\in$	*Err*	=	(*error messages*)

Domain Operations:

$empty_{\sigma} : State$
$empty_{\sigma} = \lambda\alpha.Unknown\ in\ SVal$
$New : Int \rightarrow Kont \rightarrow Cont$ [*implementation dependent*]
$Wrong : Err \rightarrow Cont$ [*implementation dependent*]
$Halt : Cont$
$Halt = \lambda\sigma.Ok\ in\ Answer$
$Prelude_{\rho} : Env$ [*implementation dependent*]
$Fetch : Loc \rightarrow Kont \rightarrow Cont$
$Fetch\ \alpha\kappa = \lambda\sigma.let\ \beta = \sigma\alpha\ in\ \beta \in Int \rightarrow \kappa(\beta \mid Int)\sigma \parallel Wrong\ ''unassigned''\ \sigma$
$Update : Loc \rightarrow EVal \rightarrow Cont \rightarrow Cont$
$Update\ \alpha\varepsilon\theta = \lambda\sigma.\theta(\sigma[\varepsilon/\alpha])$
$Unspecified : EVal$ [*implementation dependent*]

Semantic Equations:

R : $Exp \to Env \to Kont \to Cont$
P : $Cmd \to Env \to Cont \to Cont$
D : $Decl \to Env \to Dcont \to Cont$
M : $Prog \to Answer$

$$
\begin{array}{lcl}
R[\![N]\!]\rho\kappa & = & \kappa[\![N]\!] \\
R[\![I]\!]\rho\kappa & = & \textit{let } \delta=\rho[\![I]\!] \textit{ in } \delta\downarrow Loc \to \textit{Fetch}(\delta|Loc)\kappa \;[\!]\; \textit{Wrong ``not an r-value''} \\
R[\![I(E)]\!]\rho\kappa & = & \textit{let } \delta=\rho[\![I]\!] \textit{ in } \delta\downarrow Func \to \textit{Wrong ``not a function''} \\
 & & \qquad [\!]\; R[\![E]\!]\rho\{(\delta|Func)\kappa\} \\
R[\![E_1 \Omega E_2]\!]\rho\kappa & = & R[\![E_1]\!]\rho\{\lambda\varepsilon_1.R[\![E_2]\!]\rho\{\lambda\varepsilon_2.\textit{Binary}\,[\![\Omega]\!]\varepsilon_1\varepsilon_2\kappa\}\} \\
 & & \\
P[\![I:=E]\!]\rho\theta & = & \textit{let } \delta=\rho[\![I]\!] \textit{ in } \delta\downarrow Loc \to \textit{Wrong ``not an l-value''} \\
 & & \qquad [\!]\; R[\![E]\!]\rho\{\lambda\varepsilon.\textit{Update}\,(\delta|Loc)\varepsilon\theta\} \\
P[\![\text{if E } \Gamma_1\ \Gamma_2]\!]\rho\theta & = & R[\![E]\!]\rho\{Cond(P[\![\Gamma_1]\!]\rho\theta)(P[\![\Gamma_2]\!]\rho\theta)\} \\
P[\![\text{while E } \Gamma]\!]\rho\theta & = & \textit{fix}\,(\lambda\theta'.R[\![E]\!]\rho\{Cond(P[\![\Gamma]\!]\rho\theta')\theta\}) \\
P[\![\text{skip}]\!]\rho\theta & = & \theta \\
P[\![\text{return E}]\!]\rho\theta & = & \textit{let } \delta=\rho[\![\text{return}]\!] \textit{ in } \delta\downarrow Kont \to \textit{Wrong ``not in a function''} \\
 & & \qquad [\!]\; R[\![E]\!]\rho\{\delta|Kont\} \\
P[\![\Gamma_1;\Gamma_2]\!]\rho\theta & = & P[\![\Gamma_1]\!]\rho\{P[\![\Gamma_2]\!]\rho\theta\} \\
P[\![\Delta;\Gamma]\!]\rho\theta & = & D[\![\Delta]\!]\rho\{\lambda\rho'.P[\![\Gamma]\!]\rho'\theta\} \\
 & & \\
D[\![\text{int } I_1(\text{int } I_2)\Gamma]\!]\rho\chi & = & \chi(\textit{fix}\,(\lambda\rho'.\rho[(\textit{UserFunc}\,[\![I_2]\!][\![\Gamma]\!]\rho' \textit{ in } DVal)/[\![I_1]\!]])) \\
D[\![\text{int } I]\!]\rho\chi & = & \textit{New}\,\{\lambda\varepsilon.\chi(\rho[((\varepsilon \textit{ as } Loc) \textit{ in } DVal)/[\![I]\!]])\} \\
D[\![\Delta_1\ \Delta_2]\!]\rho\chi & = & D[\![\Delta_1]\!]\rho\{\lambda\rho'.D[\![\Delta_2]\!]\rho'\chi\} \\
 & & \\
M[\![\Delta;\Gamma]\!] & = & D[\![\Delta]\!](\textit{Prelude}_\rho)\{\lambda\rho.P[\![\Gamma]\!]\rho\{\textit{Halt}\}\}(\textit{empty}_\sigma)
\end{array}
$$

Auxiliary functions:

$UserFunc : Ide \to Cmd \to Env \to Func$
$UserFunc\,[\![I]\!][\![\Gamma]\!]\rho = \lambda\kappa.\lambda\varepsilon_1.\textit{New}\,\{\lambda\varepsilon_2.\textit{Update}\,\varepsilon_2\varepsilon_1\{P[\![\Gamma]\!]\rho'\{\kappa(\textit{Unspecified})\}\}\}$
$\qquad\qquad$ *where* $\rho'=\rho[(\kappa \textit{ in } DVal)/[\![\text{return}]\!][((\varepsilon_2 \textit{ as } Loc) \textit{ in } DVal)/[\![I]\!]]$

$Binary : Ops \to EVal \to EVal \to Kont \to Cont$
$Binary\,[\![+]\!]\varepsilon_1\varepsilon_2\kappa = \kappa(\varepsilon_1+\varepsilon_2)$
(*similarly for* $-$, $\times$ *and* $+$)
$Binary\,[\![<]\!]\varepsilon_1\varepsilon_2\kappa = \kappa(\varepsilon_1<\varepsilon_2 \to 1\;[\!]\;0)$
(*similarly for* $\le$, $=$, $\ne$, $\ge$ *and* $>$)

$Cond : Cont \to Cont \to Kont$
$Cond\,\theta_1\theta_2 = \lambda\varepsilon.\varepsilon\ne 0 \to \theta_1\;[\!]\;\theta_2$

Appendix B: Compiling Semantics For TINY-C

Semantic Domains:

v	∈	*Int*	=	$\{\cdots,-2,-1,0,1,2,\cdots\}^{\circ}$
I	∈	*Ide*	=	(*identifiers*)
		LocRep	=	*Ide*
		Void	=	$\{Unknown\}^{\circ}$
δ	∈	*DVal*	=	*LocRep+FuncRep+Void*
χ	∈	*Dcont*	=	*Env*→*ContRep*
ρ	∈	*Env*	=	*Ide*→*DVal*
X	∈	*Err*	=	(*error messages*)

Semantic Equations:

R	:	*Exp*→*Env*→*KontRep*→*ContRep*
P	:	*Cmd*→*Env*→*ContRep*→*ContRep*
D	:	*Decl*→*Env*→*Dcont*→*ContRep*
M	:	*Prog*→*ContRep*

$$
\begin{array}{lcl}
R[\![N]\!]\rho\kappa & = & \mathit{Literal}\ [\![N]\!]\ \kappa \\
R[\![I]\!]\rho\kappa & = & \mathit{Fetch}\ (((\rho[\![I]\!])|\mathrm{LocRep})\ \mathit{as}\ \mathit{Ide})\ \kappa \\
R[\![I(E)]\!]\rho\kappa & = & R[\![E]\!]\rho(\Lambda I_1.\ \mathit{Call}\ ((\rho[\![I]\!])|\mathrm{FuncRep})\ I_1\ \kappa) \\
R[\![E_1 \Omega E_2]\!]\rho\kappa & = & R[\![E_1]\!]\rho(\Lambda I_1.\ R[\![E_2]\!]\rho(\Lambda I_2.\ \mathit{Binary}\ [\![\Omega]\!]\ I_1\ I_2\ \kappa))
\end{array}
$$

$$
\begin{array}{lcl}
P[\![I:=E]\!]\rho\theta & = & R[\![E]\!]\rho(\Lambda I_1.\ \mathit{Update}\ (((\rho[\![I]\!])|\mathrm{LocRep})\ \mathit{as}\ \mathit{Ide})\ I_1\ 0) \\
P[\![\text{if E } \Gamma_1\ \Gamma_2]\!]\rho\theta & = & R[\![E]\!]\rho(\Lambda I.\ \mathit{Cond}\ I\ (P[\![\Gamma_1]\!]\rho\theta)\ (P[\![\Gamma_2]\!]\rho\theta)) \\
P[\![\text{while E } \Gamma]\!]\rho\theta & = & \mathit{fix}(\lambda\theta'.R[\![E]\!]\rho(\Lambda I.\ \mathit{Cond}\ I\ (P[\![\Gamma]\!]\rho\theta')\ \theta)) \\
P[\![\text{skip}]\!]\rho\theta & = & \theta \\
P[\![\text{return E}]\!]\rho\theta & = & R[\![E]\!]\rho(\Lambda I.\ \mathit{Return}\ I) \\
P[\![\Gamma_1;\Gamma_2]\!]\rho\theta & = & P[\![\Gamma_1]\!]\rho(P[\![\Gamma_2]\!]\rho\theta) \\
P[\![\Delta;\Gamma]\!]\rho\theta & = & D[\![\Delta]\!]\rho\{\lambda\rho'.P[\![\Gamma]\!]\rho'\theta\}
\end{array}
$$

$$
\begin{array}{lcl}
D[\![\text{int } I_1(I_2)\Gamma]\!]\rho\chi & = & \chi(\mathit{fix}(\lambda\rho'.\rho[(\mathit{UserFunc}\ [\![I_2]\!][\![\Gamma]\!]\rho'\ \mathit{in}\ \mathit{DVal})/[\![I_1]\!]])) \\
D[\![\text{int I}]\!]\rho\chi & = & \mathit{New}\ (\Lambda I_1.\ \chi(\rho[((I_1\ \mathit{as}\ \mathit{LocRep})\ \mathit{in}\ \mathit{DVal})/[\![I]\!]])) \\
D[\![\Delta_1\ \Delta_2]\!]\rho\chi & = & D[\![\Delta_1]\!]\rho\{\lambda\rho'.D[\![\Delta_2]\!]\rho'\chi\}
\end{array}
$$

$$
M[\![\Delta;\Gamma]\!] = D[\![\Delta]\!](\mathit{Prelude}_{\rho})\{\lambda\rho.P[\![\Gamma]\!]\rho(\mathit{Halt})\}
$$

Auxiliary functions:

UserFunc : *Ide*→*Cmd*→*Env*→*FuncRep*

$$
\begin{array}{l}
\mathit{UserFunc}\ [\![I]\!][\![\Gamma]\!]\rho = \\
\quad \mathit{MakeFunc}\ (\Lambda I_1.\ \mathit{New}\ (\Lambda I_2.\ \mathit{Update}\ I_2\ I_1\ (P[\![\Gamma]\!]\rho'(\mathit{Literal}\ \mathit{Unspecified}\ (\Lambda I_3.\ \mathit{Return}\ I_3)))))) \\
\qquad\qquad \mathit{where}\ \rho'=\rho[((I_2\ \mathit{as}\ \mathit{LocRep})\ \mathit{in}\ \mathit{DVal})/[\![I]\!]]
\end{array}
$$

MakeFunc : *KontRep*→*FuncRep*

[*extracts all temporaries to a sequence and makes a tuple of it and the body*]

$\mathit{Prelude}_{\rho}$: *Env* [*implementation dependent*]

Appendix C: Machine Semantics

Semantic Domains:

$$
\begin{array}{lcll}
 & & \mathit{Answer} & = \{\mathit{Ok}\}^{o}+\mathit{Err} \\
\alpha & \in & \mathit{Loc} & = \mathit{Int} \\
\varepsilon & \in & \mathit{EVal} & = \mathit{Int} \\
\sigma & \in & \mathit{Store} & = \mathit{Loc}\rightarrow\mathit{EVal} \\
s & \in & \mathit{Slice} & = (\mathit{Ide}\times\mathit{EVal})^{*} \\
\phi & \in & \mathit{FuncRep} & = \mathit{Ide}^{*}\times\mathit{KontRep} \\
r^{*} & \in & \mathit{RetStk} & = (\mathit{Slice}\times\mathit{KontRep})^{*} \\
\rho & \in & \mathit{Env} & = \mathit{Ide}\rightarrow\mathit{EVal} \\
 & & \mathit{State} & = \mathit{Store}\times\mathit{RetStk}\times\mathit{Env}
\end{array}
$$

Representation Syntax:

θ : ContRep (command continuations)
κ : KontRep (expression continuations)

$$
\begin{array}{lcl}
\kappa & ::= & \Lambda\mathrm{I}.\,\theta \\
\theta & ::= & \mathit{New}\ \kappa \mid \mathit{Halt} \mid \mathit{Cond}\ \mathrm{I}\ \theta_1\ \theta_2 \mid \mathit{Literal}\ \nu\ \kappa \mid \mathit{Call}\ \phi\ \mathrm{I}\ \kappa \\
 & \mid & \mathit{Return}\ \mathrm{I} \mid \mathit{Binary}\ \Omega\ \mathrm{I}_1\ \mathrm{I}_2\ \kappa \mid \mathit{Update}\ \mathrm{I}_1\ \mathrm{I}_2\ \theta \mid \mathit{Fetch}\ \mathrm{I}\ \kappa
\end{array}
$$

Execution Semantics:

Exec : $\mathit{ContRep}\times\mathit{State}\rightarrow\mathit{Answer}$
Binary : $\Omega\times\mathit{EVal}\times\mathit{EVal}\rightarrow\mathit{EVal}$

$$
\begin{array}{lcl}
\mathrm{Exec}[\![\mathit{New}\ (\Lambda\mathrm{I}.\theta)]\!]\langle\sigma,r^{*},\rho\rangle & = & \mathit{let}\ \langle\varepsilon,\sigma'\rangle=\mathit{New}\,\sigma\ \mathit{in}\ \mathrm{Exec}[\![\theta]\!]\langle\sigma',r^{*},\rho[\varepsilon/\mathrm{I}]\rangle \\
\mathrm{Exec}[\![\mathit{Halt}]\!]\langle\sigma,r^{*},\rho\rangle & = & \mathit{Ok}\ \mathit{in}\ \mathit{Answer} \\
\mathrm{Exec}[\![\mathit{Cond}\ \mathrm{I}\ \theta_1\ \theta_2]\!]\langle\sigma,r^{*},\rho\rangle & = & \rho\mathrm{I}\neq 0\rightarrow \mathrm{Exec}[\![\theta_1]\!]\langle\sigma,r^{*},\rho\rangle \parallel \mathrm{Exec}[\![\theta_2]\!]\langle\sigma,r^{*},\rho\rangle \\
\mathrm{Exec}[\![\mathit{Literal}\ \nu\ (\Lambda\mathrm{I}.\theta)]\!]\langle\sigma,r^{*},\rho\rangle & = & \mathrm{Exec}[\![\theta]\!]\langle\sigma,r^{*},\rho[\nu/\mathrm{I}]\rangle \\
\mathrm{Exec}[\![\mathit{Call}\ \langle\mathrm{I}^{*},(\Lambda\mathrm{I}_1.\theta)\rangle\ \mathrm{I}_2\ \kappa]\!]\langle\sigma,r^{*},\rho\rangle & = & \mathrm{Exec}[\![\theta]\!]\langle\sigma,\langle\mathit{mkSlice}\,\mathrm{I}^{*}\,\rho,\kappa\rangle::r^{*},\rho[(\rho\mathrm{I}_2)/\mathrm{I}_1]\rangle \\
\mathrm{Exec}[\![\mathit{Return}\ \mathrm{I}_1]\!]\langle\sigma,\langle s,(\Lambda\mathrm{I}_2.\theta)\rangle::r^{*},\rho\rangle & = & \mathrm{Exec}[\![\theta]\!]\langle\sigma,r^{*},(\mathit{unSlice}\,(s)\rho)[(\rho\mathrm{I}_1)/\mathrm{I}_2]\rangle \\
\mathrm{Exec}[\![\mathit{Binary}\ \Omega\ \mathrm{I}_1\ \mathrm{I}_2\ (\Lambda\mathrm{I}_3.\theta)]\!]\langle\sigma,r^{*},\rho\rangle & = & \mathrm{Exec}[\![\theta]\!]\langle\sigma,r^{*},\rho[(\mathit{Binary}\,\Omega(\rho\mathrm{I}_1)(\rho\mathrm{I}_2))/\mathrm{I}_3]\rangle \\
\mathrm{Exec}[\![\mathit{Update}\ \mathrm{I}_1\ \mathrm{I}_2\ \theta]\!]\langle\sigma,r^{*},\rho\rangle & = & \mathrm{Exec}[\![\theta]\!]\langle\sigma[(\rho\mathrm{I}_2)/(\rho\mathrm{I}_1)],r^{*},\rho\rangle \\
\mathrm{Exec}[\![\mathit{Fetch}\ \mathrm{I}_1\ (\Lambda\mathrm{I}_2.\theta)]\!]\langle\sigma,r^{*},\rho\rangle & = & \mathrm{Exec}[\![\theta]\!]\langle\sigma,r^{*},\rho[(\sigma(\rho\mathrm{I}_1))/\mathrm{I}_2]\rangle
\end{array}
$$

$$
\begin{array}{lcl}
\mathrm{Binary}[\![+]\!]\varepsilon_1\varepsilon_2 & = & \varepsilon_1+\varepsilon_2 \\
\text{etc.} & &
\end{array}
$$

Auxiliary Functions:

New : $\mathit{Store}\rightarrow\mathit{Loc}\times\mathit{Store}$ [*implementation dependent*]
mkSlice : $\mathit{Ide}^{*}\rightarrow\mathit{Env}\rightarrow\mathit{Slice}$
$\mathit{mkSlice}\ \langle\rangle\rho = \langle\rangle$
$\mathit{mkSlice}\ (\mathrm{I}::\mathrm{I}^{*})\rho = (\mathrm{I},\rho\mathrm{I})::(\mathit{mkSlice}\,\mathrm{I}^{*}\,\rho)$
unSlice : $\mathit{Slice}\rightarrow\mathit{Env}\rightarrow\mathit{Env}$
$\mathit{unSlice}\ \langle\rangle\rho = \rho$
$\mathit{unSlice}\ ((\mathrm{I},\varepsilon)::s)\rho = \mathit{unSlice}\ (s)(\rho[\varepsilon/\mathrm{I}])$