Semantics-Directed Code Generation

*Andrew W. Appel**

Computer Science Department
Carnegie-Mellon University
Pittsburgh, PA 15213

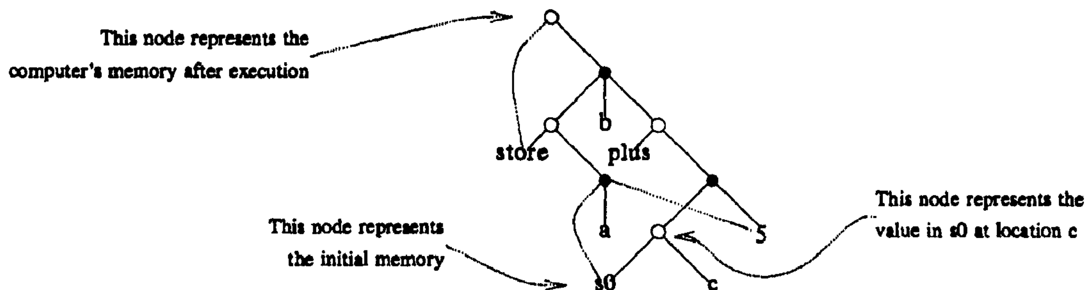
1. Introduction

The intermediate representations (IR) used by most compilers have an operational semantics. The nodes in the graph (or tree, or quad-code sequence) have an interpretation as the operation codes of some abstract machine, which may or may not be closely related to the target machine.

A denotational semantics for an IR graph, in which each node has a static meaning, can lead to a clean interface between the front and back ends of the compiler. The correctness of the front end can be checked against the semantics of the IR, and so can the correctness of the back end.

This paper describes semantics-directed compilers for Pascal and C that generate register-transfer code from such an IR graph. Code generation is accomplished by a sequence of transformations on the graph. Each transformation replaces a subgraph matching a particular pattern by a (usually) smaller subgraph, and may emit a machine-instruction; at each stage the graph continues to have a static interpretation. As in a denotational semantics for a programming language, states are represented explicitly (as internal nodes in the graph), and there are no side-effects implicit in the graph.

To illustrate, we might wish to generate machine code to implement the statement $b := c + (a := 5)$. This is represented semantically as shown (open circles represent function application; dark circles are *tuples*):

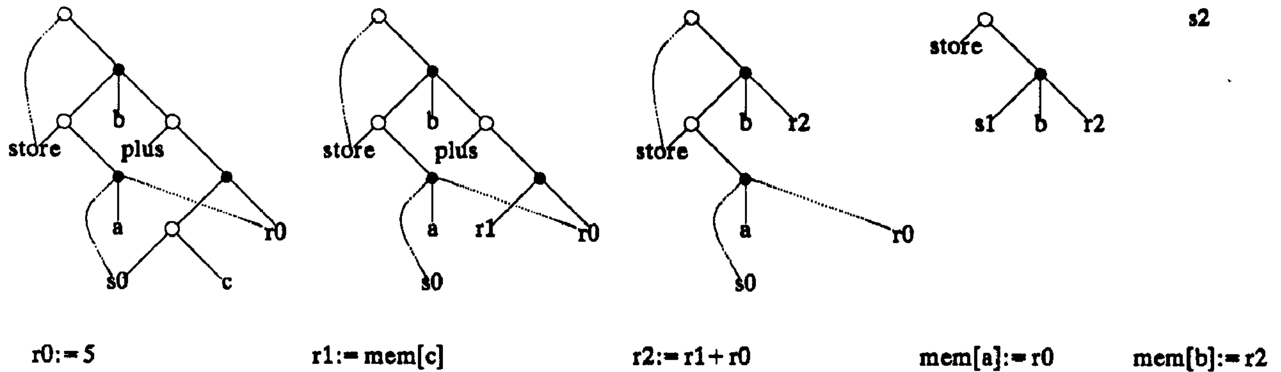


Code generation proceeds by successive reductions. Each transformation corresponds to a machine-operation; the reducer emits one line of assembly code as it performs each reduction:

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

©1984 ACM 0-89791-147-4/85/001/0315 $00.75

* Part of this work was done at AT&T Bell Laboratories, Murray Hill, NJ; part was supported by an NSF Graduate Student Fellowship.



Each transformation replaces a node in the graph by another node in the same semantic domain, and may formally specified and semantically justified.

Note that s_1 is a *different* machine memory from s_0, even if it differs only at location a. The semantic expression-graph calls for using the value of c in state s_0, not in any other state. The expression-graph may be evaluated without regard to hidden side effects; side-effects of expressions in the source language been made explicit in the IR by representing them as functions from states to states.

2. A semantics-directed compiler

Denotational semantics [10] is a technique for describing the meanings of programming-language constructs. The mathematical structures produced by a denotational semantics have a static interpretation as functions from inputs to outputs; they do not require a particular model of computation for their interpretation. (The semantic expressions may be thought of as using the λ-calculus as their model of computation. The λ-calculus has the advantage that it is free of side-effects; individual sub-expressions therefore have a well-defined, static meaning independent of their context.)

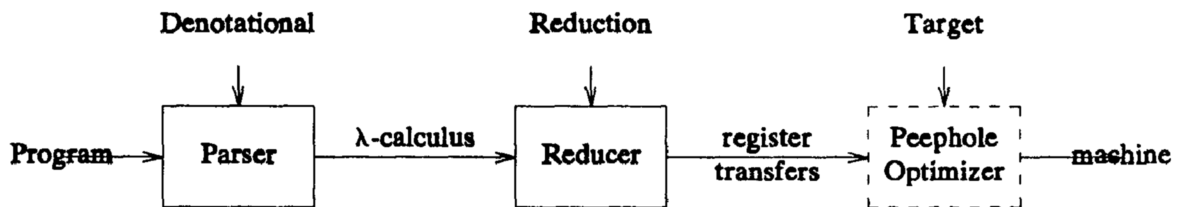
In contrast, the intermediate representations typically used by compiler-builders have usually been in the form of machine-code for some abstract machine. Sometimes the abstract machine is a "three-address" machine[6]; sometimes it is a stack-machine[12]. These compilers all have a "black box" -- with only an informal semantics -- between the parser and the IR. Some compilers use the parse-tree as an intermediate representation [14,2]; this begs the question of the semantic translation into the the IR by simply putting the black box between the IR and the emitted code.

The advantage of using an intermediate representation with a static interpretation is that the semantics of the language being compiled may be specified and discussed independently of the semantics of the target machine (concrete or abstract). Recent work in semantics-directed compilation has shown the feasibility of automatically generating translators from the source language (such as Pascal) to the λ-calculus [8,9], so that no "black box" is necessary between the parser and the IR.

Sethi [11] showed that by writing the denotational semantics of the source language in terms of a specified set of combinators, a code-generator could be made to translate the resulting λ-expressions into stack-machine code. However, his choice of combinators was very restrictive, and his reducer was not sufficiently powerful to compile languages with the varied data types of Pascal-like languages. Wand[13] gives a set of combinators similar for stack-machine interpretation, but again the semantic specification is severely restricted and the combinators are not sufficiently close to von Neumann machine-instructions.

This paper describes a semantics-directed compiler-generator which is powerful enough to implement compilers for Pascal and C. The front end of the compiler takes the program source as input and produces, as output, a λ-expression representing the denotation of the program. The reducer takes this expression and produces register-transfer "assembly code." This register-transfer code is designed to be fed to fed to a register allocator and peephole optimizer phase, although this phase has not been implemented. None of the phases in this compiler is a "black box." All have

their semantics described formally, and each is produced automatically from its formal description.



The front end is generated automatically from a denotational-semantic description. Such front ends have been described in [8] , [9] , and [11]; no detailed description is given here.

The peephole optimizer may be generated automatically from the target-machine description, as in [3] , [4], and [7]; it performs the task of assigning temporaries to registers and of choosing appropriate target-machine instructions to effect the specified register transfers.

This paper describes the reducer. Section 3 briefly summarizes the semantic language used in the front end; section 4 describes the notation for specifying reductions to the reducer-generator. Sections 5, 6, and 7 describe the reductions which are useful for compiling conventional programming languages. Section 8 discusses the algorithm for applying these reductions. Section 9 discusses the compilation of languages with structured data types, and the last section gives an overview of compiler performance and open problems.

3. Representation of expressions

The front end produces a λ-calculus expression representing the meaning of the source program. Expressions are represented as directed graphs, rather than trees, to permit sharing of common sub-expressions. The sharing of common sub-expressions is extremely important, because the domains of expressions include machine-states, which must not be copied by the reducer; and continuations, which if needlessly copied will produce needless duplication of emitted code.

The syntax and semantics of λ-expressions is very similar to that of ML[5], and is summarized as follows:

identifier	Variable identifiers and combinators.
λ *x* . *expr*	Lambda-abstraction on the variable *x*
expr expr	Application of first *expr* to second.
(*expr*, . . . ,*expr*)	N-tuples
expr.i	Selection of ith element from N-tuple*

Tagged variants as in Hope [1] are also used.

Standard reductions are associated with these operators. A selection node connected to a tuple node reduces by selecting one of the elements; a λ-node applied to an argument reduces by β-reduction.

4. User-specified reductions

In addition, the compiler-writer may introduce arbitrary combinator symbols, and provide reductions associated with them. This differs from simple λ-abstraction because several reductions may be provided for the same combinator. These may correspond to different ways of implementing the same thing; for example, a reduction for the combinator *plus* might emit an "add" instruction, while a different reduction (applicable only if both arguments have been evaluated to integers) could evaluate the addition at compile-time.

* This notation for selection is not used in the semantic language; there, selection is accomplished by lambda-binding a tuple, as in λ (*x*,*y*,*z*) . *plus*(*x*,*plus*(*y*,*z*)). In the graph representation, however, these are converted to "select-i^{th}-element" nodes.

Reductions are specified as "pattern → substitution." The language of patterns is similar to a restricted form of the language of expressions:

identifier	Variable name.
identifier : identifier	Variable name with representation specification.
identifier	Constant-symbol or combinator.
pattern pattern	Application of first *pattern* to second.
(*pattern, . . . ,pattern*)	N-tuple

We will use the pattern **plus** (*a:Num*, *b:Num*) to illustrate the semantics of patterns. This pattern specifies the combinator *plus* applied to a pair (2-tuple) of numbers.

In the denotational-semantic specification, *plus* is declared to be in the domain $(Int \times Int) \rightarrow Int$. It would seem from the domain declaration that *plus* could not be applied to anything *but* two numbers; however, *plus* may be applied to any pair of *expressions* that evaluate to something in the domain *Int*. The pattern above will only be matched when these two expressions have been fully evaluated (i.e., when *a* is represented as a numeric constant -- as specified by ":*Num*" -- rather than as a more general expression).

The right-hand-side of a reduction specification may be another pattern, or it may be specified in the C language. A register-transfer statement to be emitted may also be specified.

4.1. Substitions written in C

The C-language substitutions are useful chiefly for implementing the constant-folding of operators about which the λ-machine has no need of knowing. The compile-time evaluation of additions is accomplished thus:

plus (*a:Num*, *b:Num*) → { return number(a+b); }

where *number* is a C function that returns an expression node representing a numeric constant. Using this reduction, the expression *plus*(*plus*(3,4),8) could be reduced in two steps to (15).

4.2. Substitions written as patterns

Substitutions may also be specified as patterns. The same syntax applies; variables in the substitution that also appear in the left-hand-side specify the re-use of the corresponding nodes.

A reduction to simplify an expression containing addition and negation is specified by

plus (*a*, **negate** *b*) → **minus** (*a*, *b*);

Note that *a* and *b* have no representation specification here, as this reduction is applicable regardless of their representation.

4.3. Code emission

When the substitution is a pattern, a line of register-transfer code may be emitted as the reduction is done. This line may use any of the variables used in either side of the reduction. (Variables in the substitution that do not appear in the right-hand-side must have a representation specifier, and indicate the use of a "new" node of that type). Here is a reduction that emits an "add" instruction:

plus (*a:Reg*, *b:Reg*) → *c:Reg* "c := a+b"

Thus, *plus*(r_6,r_7) could be replaced by r_9, while emitting the instruction "r9 := r6+r7."

The *store* combinator performs a role similar to that of the "store" machine-instruction of a von Neumann machine. That is, *store*(*s*,*l*,*v*) produces a new state *s'* similar to the state *s*, except that at the location* *l* it produces the value *v*. This is expressed straightforwardly as a reduction:

store (*s:State*, *l:Ide*, *v:Reg*) → s_1:*State* "mem[*l*] := *v*"

5. A simple example

A denotational semantics for a simple expression-language is given, with a set of reductions sufficient to generate assembly code.

5.1. Domain declarations

$S = Ide \to V$	**States are functions from Identifiers to Values.***
$[\text{expr}]: S \to (V \times S)$	**An expression, given a state, produces a value and a new state (which is similar to the old state but may have been side-effected). expr is a syntactic unit; [expr] is its denotation.**
$[\text{goal}]: S$	**A sentence in the language denotes the state resulting from evaluating the expression.**
$[\text{ID}]: Ide$	**The terminal symbol ID has an identifier as its denotation.**
$[\text{NUM}]: V$	**The symbol NUM has a value as its denotation.**
$s_0: S$	**The initial state.**
$plus: (V \times V) \to V$	***plus* is a dyadic function on the domain of values.**
$store: (S \times Ide \times V) \to S$	**As described in section 4.3**

5.2. Denotational semantics

goal → expr	let $s,v = [\text{expr}]\ s_0$ in s
expr → NUM	$\lambda s.([\text{NUM}], s)$
expr → ID	$\lambda s.(s\ [\text{ID}], s)$
expr → expr + expr	$\lambda s.$ let $s_1,v_1 = [\text{expr}_1]\ s$ in let $s_2,v_2 = [\text{expr}_2]\ s_1$ in $(plus(v_1,v_2), s_2)$
expr → ID := expr	$\lambda s.$ let $v = [\text{expr}]\ s$ in $(v, store(s,[\text{ID}],v))$
expr → expr ; expr	$\lambda s.$ let $s_1,v_1 = [\text{expr}_1]\ s$ in $[\text{expr}_2]\ s_1$
expr → (expr)	$[\text{expr}]$

5.3. Reductions

plus $(a{:}Reg,\ b{:}Reg) \to c{:}Reg$	"$c := a+b$"
$i{:}Num \to r{:}Reg$	"$r := i$"
$s{:}State\ a{:}Ide \to r{:}Reg$	"$r :=$ mem$[a]$"
store $(s{:}State, a{:}Ide, v{:}Reg) \to s_1{:}State$	"mem$[a] := v$"

This reducer will work correctly only if there is at most one node in the graph with the *State* representation at any given time, just as von Neumann machines may have only one state at a given time. To ensure this condition, the last reduction may never be used while there is any instance in the expression-graph of the pattern on the left-hand-side of the third reduction; this intuitively means that any reading from a state must be done before that state is destroyed (this will be discussed later in more detail).

5.4. A compilation

Using this language specification and reducer, we may translate the sentence

$$b := c + (a := 5)$$

After all β-reduction and tuple-selection has been done, the expression-graph shown in section 1 remains. The reduction-sequence shown in section 1 yields the register-transfer code:

```
r0 := 5
r1 := mem[c]
r2 := r1 + r0
```

* For the purposes of this explication, addresses are represented as identifiers, so that $S = Ide \to V$; in an actual compiler, addresses are themselves values.

mem[a] := r0
mem[b] := r2

The same reducer might choose to apply the reductions in a different order to generate slightly different code. For example, mem[c] might be fetched before 5 is loaded.

Although the semantic definition seems to specify left-to-right evaluation, in evaluating the arguments of the *plus* the same result will be obtained either way. This is because all side-effects have been made explicit by the semantics; the semantic graph does not require any particular evaluation order. Different reduction orders will often lead to different patterns of register usage, and a sophisticated reducer might make its choices so as to reduce the number of registers needed.

6. Continuations

The domain $C=S \rightarrow A$ of continuations — functions from states to answers — is useful in describing the semantics of conventional programming languages. A continuation may be written in the λ-calculus in the obvious way — by a λ-function whose bound variable is in the domain S. The application of a continuation to a state may be reduced by β-reduction. No special machinery is required.

Unfortunately, β-reduction often requires copying the λ-expression being applied; this is necessary when the λ-expression is also applied to some other argument in a different part of the graph. (Recall that common subexpressions are shared in the graph representation. If they were not, then copying of the *argument* would be required if the bound variable occurred more than once in the body of the λ-function.)

The copying of a subexpression implies that machine-code will be generated for each copy. The application of a continuation c to a state s is therefore to be avoided when there are instances in the graph of $c\ s'$ for $s' \neq s$. What will be done is to take advantage of the intuitive correspondence between continuations and assembly-language statement-labels.

Given a continuation c of the form $\lambda s.answer$, we may assume that code could be generated to implement that continuation, starting at some label L. The continuation c is replaced in the graph by a "goto"; instead of β-reducing $c\ s_0$, we emit the instruction "goto L." The reductions to accomplish this are as follows:

c:*Cont* → **keep**(**genbody**(*c*,*l*:*Label*))(**goto** *l*)

goto *l*:*Label* *s*:*State* → **undefined** "goto *l*"

genbody(*c*:*Cont*, *l*:*Label*) → *c* *s*:*State* "*l*:"

The **keep** combinator actually splits the graph into two graphs (which may, however, share common subexpressions). If we start with the expression

$$(\cdots (\textbf{keep}\ e_2\ x) \cdots)$$

then after **keep** is reduced, two expressions will remain:

$$(\cdots (x) \cdots) \quad \text{and} \quad e_2$$

Just as there was a restriction on the applicability of the *store* combinator in the previous section (i.e, that it could not be used if the state s was still being used elsewhere in the graph), there are restrictions on the applicability of these reductions:

The first of these reductions — which chops a subgraph out of the graph — should not be used if the continuation c contains free variables. It turns out that if c is (somewhere in the graph) applied to the current state (i.e., a node having the *State* representation), then c will have no free variables.

The second reduction should not be used if there are other references to the state s. (This restriction is identical to the restriction on *store*.)

The last of these reductions — which begins a new "basic block" — should not be used if the reducer is still in the middle of generating code for some other basic block. This condition

will be satisfied by simple avoiding this reduction as long as there is some node in the graph with the *State* representation. (Note that the reduction that emits a "goto" also has the effect of removing the last reference to a *State* node; this ends a basic block.)

This leaves the problem of determining which λ-nodes are continuations (i.e., given a node s, determining whether it is in the representation-class *Cont*). If the front-end (which processes the semantic description) has a type-checker (which is recommended), then it can mark all λ-nodes in the domain C. The part of the reducer that does β-reduction must not β-reduce λ-nodes with this mark; instead, it should leave them to be handled by the reductions in this section.

7. Conditionals

With the reductions to handle continuations already in place, condition-operators are simple. Again, to find the appropriate semantic domain, consider the intuitive semantics of a typical conditional-goto machine instruction. Two values are compared; depending on whether they meet the test (i.e. equality), either a label is branched to, or the next statement is the continuation.

In the last section the correspondence between labels and the domain C was described. The domain of the **eq** combinator is **eq**: $(V \times V \times S \times C \times C) \rightarrow A$. The interpretation of $eq(v_1, v_2, s, c_1, c_2)$ is that if v_1 is equal to v_2, then this is equivalent to $c_1\ s$; if not equal, to $c_2\ s$. In fact, the reduction to perform constant-folding of *eq* expresses this as follows:

$$\mathbf{eq}(v_1{:}Num, v_2{:}Num, s, c_1, c_2) \rightarrow$$ {return apply(v1==v2?c1:c2, s);}

To generate code for a conditional, we use the fact that c_1 can be represented as a label:

$$op{:}Comp\ (v_1{:}Reg, v_2{:}Reg, s{:}State, (\mathbf{goto}\ l{:}Label), c_2) \rightarrow c_2\ s \qquad \text{"if } v_1\ op\ v_2 \text{ goto } l\text{"}$$

This is subject to the same restriction on other uses of s as are the *store* and *goto* reductions.

8. Ordering the reductions

The set of reductions presented thus far are sufficient (with a few adjustments) to build a compiler for a language such as Pascal or C. As the reductions were presented, however, restrictions on their use were mentioned. This section describes an algorithm to implement a reducer consistent with these restrictions; it is not necessarily the only possible algorithm.

The idea is to divide the reductions into several *classes*. The reductions in lower-numbered classes take precedence over those in higher-numbered classes. More formally, at any point in the reduction sequence, no reduction of class j may be applied if there is a reduction in class i (for $i \leq j$) that may also be applied. For example, the restriction that a state may not be updated until there are no pending "fetches" from that state may be implemented by putting the "fetch" reduction in a lower reduction class than the "store" reduction.

The division into classes of the reductions discussed in this paper is as follows:

The first set of reductions are the "classical" kind that emit no code.

β-reduction
tuple-selection
tag-selection
plus $(a{:}Num,\ b{:}Num) \rightarrow$ { return number(a+b); }

The second class consists of the reductions that emit code which does not change the state.

$op{:}Binop\ (a{:}Reg,\ b{:}Reg) \rightarrow c{:}Reg$	"c := a op b"
$i{:}Num \rightarrow r{:}Reg$	"r := i"
$s{:}State\ a{:}Ide \rightarrow r{:}Reg$	"r := mem[a]"

The third class of reductions change the state in some way.

store (s:*State*,a:*Ide*,v:*Reg*) → s_1:*State* "mem[a] := r"

c:*Cont* → **keep(genbody(**c,l:*Label*))(**goto** l)

goto l:*Label* s:*State* → **undefined** "goto l"

This reduction begins a new basic block.

genbody(c:*Cont*, l:*Label*) → c s:*State*"l:"

The patterns on the left-hand-sides of the reductions all have a bounded depth. This fact can be exploited to quickly find matches to these patterns. A queue U of unexamined nodes is maintained. As nodes are removed from this queue, they are matched by a pattern-matcher and put in a queue C_i corresponding to the appropriate reduction-class. When U is empty, then a reduction may be done on a node in the lowest-numbered non-empty C_i.

Performing a substitution will create new patterns in the graph. However, these are localized, so only a small number of nodes will be affected. These nodes will be put into the queue U.

Graph nodes need not be processed through U and the C_i in a first-in, first-out order (or any other particular order), since any order consistent with the class-priority restriction will produce correct code. In fact, β-reductions require some care, as some reduction orders will never terminate on some inputs, while other reduction orders *will* terminate on the same inputs. However, selecting nodes at random from the queues is a mathematically-justified way of ensuring termination on inputs that have *some* terminating reduction-sequence. (This will be true as long as the size of the graph does not diverge.)

9. Data types in Algol-like languages.

High-level languages (or rather, languages so considered in 1970) allow variables in such domains as integers, reals, records (modelled in a denotational semantics as functions from identifiers to values), and arrays (modelled as functions from integers to values). Machines, on the other hand, provide only one data type -- the "word" -- with a set of integer, real, and indexing operations upon words.

The reductions presented in this paper are suited to the implementation of functions upon words, but not of the higher-order functions provided by Algol-like languages. It is the job of the semantics to map these higher-order functions into the domain of words.

For example, the semantics for subscripting an array variable might look like this:

var → var . ID λeλs. *ckrec*(*fetch*(*[*var*]* e) s) *[*ID*]*

with the meaning, "evaluate the var in the environment e, fetch it from the state s, check that it is a record variable, and then apply the resulting *Ide*→*Lv* function to the identifier ID.

The problem is that the reducer doesn't know about records and identifiers; all it knows about are addition and states. Furthermore, if the reducer *could* be made to know how to generate code to look up identifiers in record environments, the result wouldn't be what we had in mind at all! That kind of evaluation should be done at compile time.

What should be done, then, is to describe such functions as *ckrec*, and the *Ide*→*Lv* functions representing records, in the λ-calculus, so that ordinary β-reduction can do at compile time the things that we believe compilers should do at compile-time.

In a denotational semantics, however, there is no distinction between run-time and compile-time. Indeed, the semantics itself does not even specify that the program *should* be run. Therefore, although we wish to make the compiler handle certain kinds of reductions at compile-time, and generate code to handle others, it would be nice if this division did not unduly twist the structure of the semantic definition. In this section a set of semantic definitions is provided to map the high-level data types onto the machine semantics.

Let L-values be functions from states to R-values (Lv=S→Rv), and let R-values be the union

of various domains:

$$Rv = \textbf{int of } Int \mid \textbf{real of } Real \mid \textbf{array of } Int \rightarrow Lv \mid \cdots$$

Finally, let a "type" be a mapping from a location to an L-value ($Ty = L \rightarrow Lv$). Then we can define the integer and real types:

$$int_ty = \lambda l.\lambda s.\ \textbf{int of } s\ l$$

$$real_ty = \lambda l.\lambda s.\ \textbf{real of } s\ l$$

Unfortunately, this doesn't allow for the updating of the state! What must now be done is to make things a little less abstract. Instead of L-values being functions from states to r-values, they will also have locations and sizes attached: $Lv = (S \rightarrow Rv) \times (L \times Int)$. This location and size will be available to the (typed) "update" function, which will be implemented in terms of the (untyped) "store" function.

$$int_ty = \lambda l.(\lambda s.\ \textbf{int of } s\ l),(l,1)$$

The array constructor is parameterized by element-type and number of elements:

$$make_array\ ty\ n = \lambda l.\ \text{let } f,(l',size) = ty\ l \text{ in } (\lambda s.\textbf{array of } \lambda i.ty(l + size \times i)),(l, size \times n)$$

This "breaking of the abstraction", in giving an l-value two somewhat redundant representations (on one hand, as a function from states to r-values, and on the other, as a location and a size) can be "hidden" from the high-level semantics by a proper modularization of the semantic definition.

Using this technique, the semantics behaves a lot like a "real" denotational semantics, even though there is a less-than-abstract aspect to it. Most importantly, the code generator can deal with a machine semantics rather than a high-level semantics.

10. Implementation details and further research

There is room for much improvement in this method of code generation. The compilers that use this reducer can compile at the rate of less than one line per second, and use a lot of space; and there are some problems in the way mathematically abstract functions are mapped into machine code.

10.1. Parameters of the implementation

The specification of a Pascal compiler requires approximately 700 lines of semantics, and 50 lines of reductions. The reducer as implemented is fairly slow; most of its time is spent copying nodes for β-reduction. (This copying is what causes the growth from "Initial size of graph" to "Maximum size of graph" in the table below.)

Program name	Lines of Pascal	Initial Size of graph	Maximum Size of graph	Code emitted	Reductions performed	CPU time (VAX 750)
tiny	8	1427 nodes	1600 nodes	32 lines	436	13 sec.
queens	39	3486	9500	330	5410	60
trees	57	2756	7800	322	4119	58
accel	85	4148	17200	675	11663	464

It may be possible to (automagically) process the semantics to perform more reductions at compiler-generation time (about 400 reductions are done on the present Pascal-compiler before reading the input). Or it may be that the same number of reductions may be done more efficiently by a better reducing algorithm.

10.2. Eager evaluation of fetches

The specification of the reducer calls for the application of the reductions in class 2 (which emit "add" and "fetch" instructions) before the use of those in class 3 (which emit "goto" and "store" instructions), whenever possible. This guarantees correctness, but has some unpleasant consequences. What happens is that whenever a value becomes available, it is loaded into a register. This has a very unfortunate effect on register usage. Values which are used at the end of a procedure may be loaded into registers, sitting there untouched until many instructions later when they are used.

Something even worse happens when an instruction which is modelled in the semantics as a pure function -- such as *plus*, from integers to integers -- has the capability of halting the processor, such as by overflow. A very similar example is the bounds-checking of subranges, and the testing of pointers for NIL before dereferencing. All of these can be modelled in the semantics by functions that do not refer to the state, but can generate error:

$$plus = \lambda(x,y).\ \textbf{if}\ x+y>maxint\ \textbf{then error else}\ x+y$$

The problem is that error is not simply an undefined value to be loaded into a register (as it would be in the CDC-6600); most machines will generate an overflow exception which (in Pascal) halts the execution of the program. This problem shows up in the following Pascal code:

if i<0 **then** i := i+maxint

This might be translated into machine code that loads i+maxint into a register, then does a conditional branch (one branch of which assigns the register into i). However, this will overflow if $i>0$, so the addition should be performed only *after* the conditional branch.

The solution is to use lazy evaluation of the reductions in this class. By avoiding them until necessary, values won't be loaded into registers until they are truly needed. This will aid register allocation and avoid extraneous overflows.

References

1. R. Burstall, D. MacQueen, and D. Sannella, "Hope: an Experimental Applicative Language," *Proceedings of the 1980 LISP Conference*, pp. 136-143 (1980).
2. R. G. G. Cattell, "Formalization and automatic derivation of code generators," Ph.D. Thesis, Carnegie-Mellon University, Pittsburgh, PA (April 1978).
3. J. W. Davidson, "Simplifying Code Generation Through Peephole Optimization," TR 81-19, Department of Computer Science, University of Arizona, Tucson, Arizona (1981).
4. J. W. Davidson and Christopher W. Fraser, "Automatic Generation of Peephole Optimizations," *Sigplan '84 Symposium on Compiler Construction*, pp. 111-116 ACM, (1984).
5. M. J. Gordon, A. J. Milner, and C. P. Wadsworth, *Edinburgh LCF*, Springer-Verlag, Berlin (1979).
6. IBM, "FORTRAN IV (H) compiler program logic manual," Form Y28-6642-3, IBM, New York, NY (1968).
7. Robert R. Kessler, "Peep -- An Architectural Description Driven Peephole Optimizer," *Sigplan '84 Symposium on Compiler Construction*, pp. 106-110 ACM, (1984).
8. P. D. Mosses, "SIS -- Reference and user's guide," DAIMI MD-30, Computer Science Department, University of Aarhus, Denmark (1979).
9. L. Paulson, "A Semantics-Directed Compiler Generator," *Ninth ACM Symposium on Principles of Programming Languages*, pp. 224-233 ACM, (1982).
10. D. S. Scott and C. Strachey, "Towards a mathematical semantics for computer languages," *Proc. Symp. Computers and Automata*, pp. 19-46 Polytechnic Press, (1971).
11. R. Sethi, "Control Flow Aspects of Semantics Directed Compiling," *Trans. Prog. Lang. and Systems* 5(4) pp. 554-595 ACM, (October 1983).
12. A. S. Tanenbaum, H. van Staveren, and J. W. Stevenson, "Using Peephole Optimization on Intermediate Code," *Trans. Prog. Lang. and Systems* 4(1) pp. 21-36 ACM, (1982).
13. M. Wand, "Deriving target code as a representation of continuation semantics," *ACM Trans. Programming Languages and Systems* 4(3) pp. 496-517 (July 1982).
14. W. Wulf, R. K. Johnsson, C. B. Weinstock, C. B. Hobbs, and C. M. Geschke, *Design of an Optimizing Compiler*, Elsevier North-Holland, New York (1975).