Circular expressions: elimination of static environments

Ravi Sethi

Bell Laboratories
Murray Hill, New Jersey 07974

ABSTRACT

Consider the connection between denotational semantics for a language with **goto** statements and flow diagrams for programs in such a language. The main point of interest is that the denotational semantics uses a recursively defined environment to give the meaning of labels, while a flow diagram merely has a jump to the appropriate program point. A simple reduction called "indirection elimination" strips away the environment from the denotational semantics and extracts an expression with cycles (circular expression) that is very close to the flow diagram of a program. The same idea applies to associating bodies with recursive procedures, or to any construct whose semantics is not wedded to the syntax. Circular expressions are offered as a useful data structure and conceptual device. Expressions with cycles are well defined mathematical objects — their semantics can be given by unfolding them into infinite structures that have been well studied. The practicality of the elimination of environments has been tested by constructing a trial implementation, which serves as the front end of a semantics directed compiler generator. The implementation takes a denotational semantics of a language and constructs a "black box" that maps programs in the language into an intermediate representation. The intermediate representation is a circular expression.

1. Introduction

Static environments. The term "context sensitive syntax" is sometimes used to refer to properties of a program that are evident from the syntax, but are hard to specify with a context free grammar. Speaking in broad terms, programs may contain "things" that are **defined** and **used** in ways that cannot be specified with a context free grammar. For example labels are defined where they occur on statements, and are used in **goto**'s. Similarly procedures are defined when they are declared, and are used when they are invoked. By defining and using type synonyms, a structure containing a pointer to itself can be created in some languages. Even if **goto** statements are not allowed, hidden labels might be defined and used due to **break** and **continue** statements. In each of the above examples, the **definition** of a thing may be at a point syntactically unrelated to the **use** of the thing (in a context free syntax).

A mechanism like a symbol table is therefore needed so that a **use** can locate the corresponding **definition** indirectly through the table. Such symbol tables are called *environments*. When an environment is used to connect a **definition** and **use** that are evident from the syntax, then the environment is referred to as a *static* environment. Note that at language specification time, any program is fair game, so static environments are essential to a language specification based on a context free syntax.

The above discussion can be made more concrete by considering **goto** statements and flow of control, but it applies more generally. The semantics of structured constructs like **while** statements do not depend on the context in which they appear. The meaning of such constructs can therefore be determined from the meaning of the subconstructs by providing appropriate "glue". This approach does not work for a language with **goto** statements. The standard denotational semantics for such languages [mil76] first determines an environment containing the meanings of labels, and then uses this environment to give the meaning of a block.

Semantics directed compiler generation. Now consider the problem of generating a compiler directly from the denotational semantics of a programming language. Denotational specifications of programming languages follow the convention of giving the meaning of a construct strictly in terms of its subconstructs. For example, the meaning of a **while** statement is given in terms of the meanings of the test and the body alone — not the whole **while** statement. Therefore, in a syntax directed fashion it is theoretically possible to map a program into an expression in the metalanguage that is used to specify semantics. The logical organization of a program for performing this mapping is described in Section 5. The program takes as input the semantics of a language, and constructs a "black box" to perform the syntax directed mapping.

The output of the syntax directed mapping will contain static environments and other constructions that can be eliminated "at compile time" i.e. when the syntax directed mapping is done (as opposed to when the semantics of the language are specified). Using the example of **goto** statements once again, it is shown in Section 4 that static environments can easily be eliminated, and that something close to a flow diagram for a program can be extracted from the output of the syntax directed mapping.

Programs that run on conventional computers have loops. But there are no cycles in the standard denotational semantics of programming languages [mil76]. Instead, least fixed points are used to give meanings for recursion and iteration. This difference can be bridged by introducing expressions with cycles.

Circular expressions. In an elegant mathematical treatment of flow diagrams, Scott [sco71] remarks, "There are many shortcuts one can take in the drawing of diagrams to avoid tiresome repetitions. ... Loops may just be an extreme case of abbreviation." As the remark suggests, loops make a brief appearance in [sco71] only to be unfolded into acyclic, though infinite, diagrams.

I would like to revive *circular* rather than infinite structures, both from a conceptual and from a data representation viewpoint. In the early days, Landin [lan64] talked of "circular definitions". The phrase "tying the knot", presumably to create a circular expression, has been attributed to Strachey. Infinite structures are still of interest since they are needed to specify the semantics of circular expressions: the basic idea is to unfold a circular expression into an infinite acyclic one — Appendix A contains references.

Outline. There is a close connection between least fixed points and circular expressions. Each instance of the least fixed point operator (*fix*) can be replaced by the simple circular expression in Figure 1. When a subgraph representing a function is applied to a subgraph representing an argument, some simplification can take place. (This simplification would be called β-reduction in the λ-calculus [chu41].) Figure 2 contains an example showing how such simplification results in other cycles. See Section 2 for details.

Static environments are studied abstractly in Section 3 in terms of a class of circular expressions. Under some assumptions that are expected to hold in practice, it is shown that static environments can be easily eliminated. The example of **goto** statements is considered in Section 4 to provide evidence that the assumptions are reasonable. Using circular expressions, the environment and labels are transformed away, resulting in an expression in which the cycles correspond naturally to the intended **goto** statements. Since the environment is used in a similar manner to determine the meanings of recursive procedures, circular expressions help there as well.

A trial implementation sketched in Section 5 attests to the practical utility of the development in this paper. Sample inputs and outputs are shown in Appendix B. Section 6 discusses some of the experiments that have been performed and contains brief speculation on the remaining task of mapping reduced circular expressions to something that runs.

While I have attempted to write simply and clearly, some readers may benefit from the tutorial presentations in [gor79, sto77].

2. Circular expressions versus fixed points

The first indication that circular expressions may be useful comes from a connection between the least fixed point operator (*fix*) and the graph in Figure 1. Let ***app*** represent an operator that applies its first argument to its second. So, the ***app*** in Figure 1 applies f to the expression graph rooted at ***app***. As this graph is unfolded, the sequence of expressions

$\lambda f.\ \Omega,\ \lambda f.\ f(\Omega),\ \lambda f.\ f(f(\Omega)),\ \cdots$

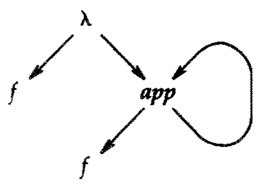

Figure 1. Let *app* represent an operator that applies its first argument to its second. When the above graph is unfolded, the infinite expression $\lambda f.\ f(f(f(\cdots)))$ is obtained. It is well known that this expression is semantically equivalent to the least fixed point operator.

is obtained, where Ω marks the place at which further unfolding needs to take place. The fully unfolded form is the infinite expression $\lambda f.\ f(f(f(\cdots)))$. It is well known [par70, sco76] that this infinite expression is semantically equivalent to the least fixed point operator.

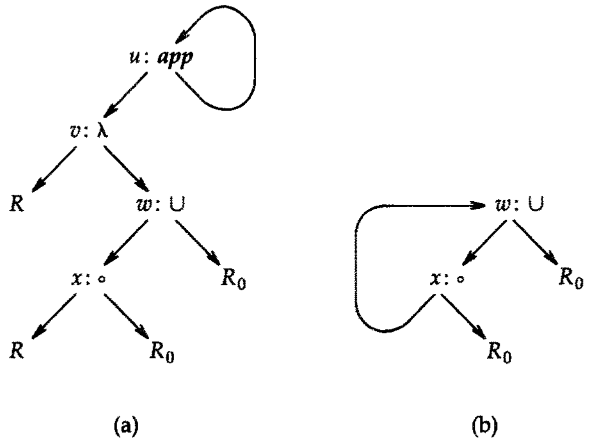

Figure 2. The root of the expression in (a) is the vertex u. Since the value of u is the result of applying the function at v to an argument, the value of u is obtained from w by substituting the argument for all instances of the bound variable R. The reduction is performed in two stages. First, all edges into the bound variable are redirected to the vertex that the function at v is applied to. In this case the edge from x to the leaf for R is redirected to u. Second, all edges into u are redirected into w which now represents the value previously represented by u. The result is the expression in (b). As an aside, "$\circ$" represents composition, "$\cup$" represents set union, and the relation being defined is the transitive closure of the binary relation R_0. This example has been taken from a paper on query languages for data bases [aho79].

Once the graph in Figure 1 is allowed, further cycles arise naturally. Consider for example:

$$fix\ (\ \lambda R.\ R \circ R_0 \cup R_0\ ) \tag{1}$$

First, replace *fix* by the graph in Figure 1 i.e. apply the function represented by the graph in Figure 1 to the argument of *fix*. β-reduction now yields the circular expression in Figure 2(a). In the λ-calculus:

$$(\lambda x.M)N \to_\beta M[N/x]$$

Here $M[N/x]$ represents the expression obtained by substituting N for all instances of x in M. With graphs, the substitution $M[N/x]$ corresponds to redirecting all edges to x into the root of N. Such a substitution rule for acyclic graphs is given in [pac74, sta79]. The details for circular expressions are a matter of routine programming. In particular, the graph in Figure 2(b) is obtained from 2(a) by β-reduction.

Drawing graphs on blackboards is one thing: instructing a text processing system to draw them is another. Instead of having to draw circular expressions all the time, it will be convenient to have a linear representation using recursive schemes. The schemes are for human consumption only — all algorithms operate on graphs.

Playing the usual game of cutting all loops and writing recursive schemes, the value at vertex w in Figure 2(b) must satisfy the equality:

$$w = w \circ R_0 \cup R_0 \qquad (2)$$

where w does double duty as a variable for a relation in this equality. The recursive scheme for a graph is not unique. With multiple, intersecting loops, there is in general a choice of cutpoints. Even the simple graph in Figure 2(b) has the alternate representation

$$w \ \textbf{\textit{where}}\ w = x \cup R_0 \ \textbf{\textit{and}}\ x = w \circ R_0$$

Fortunately it can be shown that all recursive schemes representing the same graph have the same infinite tree as their least fixed point. (Scott [sco71] uses flow diagrams for motivation alone: the mathematics deals with recursive schemes. This is why Scott has trouble writing a flow diagram for a particular recursive scheme [adj77, p.90].)

3. Elimination of static environments

In denotational semantics, the term *environment* generally refers to a function that relates a "name and the thing it denotes" [str72]. While there are no a priori restrictions, in practice, environments are used essentially as symbol tables. There may be circularity in the definition of the symbol tables since things like types can be defined in terms of themselves. This section begins with an informal discussion of some conditions to be placed on environments. The example of **goto** statements in the next section (which is self contained) suggests that the conditions are natural. Environments satisfying the conditions can be eliminated without detailed analysis of the circular expression for a program.

The conditions to be placed on static environments are a formalization of the following.

E1. An environment is either given, or is obtained by updating a static environment. Circularity is permitted e.g. a static environment may be obtained by updating itself.

E2. Environments may be used only to determine the things denoted by names.

E3. The static aspects of environments can be formalized by requiring that environments are applied only to known entities. For example, if an array `A` is indexed only by constants, as in `A[2]` or `A[23]`, then the array is used in a static manner; otherwise if `A[i]` is allowed then the static properties are lost since `i` has to be evaluated dynamically to determine what `A[i]` refers to.

In formalizing the above conditions, an operator ***upd*** will be used:

$$\textbf{\textit{upd}}(f,z,a) \equiv \lambda x.\ \textit{if}\ x=z\ \textbf{\textit{then}}\ a\ \textbf{\textit{else}}\ f(x)$$

In circular expressions, ***upd***(f,z,a) is encoded by constructing a vertex with three sons, the first representing f, the second z, and the third a. ***app***(f,x) is encoded by constructing a vertex with two sons, the first representing f and the second x.

In order to focus on environments, the only functions that will be considered are environments. Thus, all instances of the operator ***app*** correspond to the application of a static environment to a name. Similarly, all instances of the ***upd*** operator correspond to the updating of an environment. Since uninterpreted operators will also be allowed, other kinds of functions can also be applied and updated using uninterpreted operators: their presence will just not influence the discussion.

With all the above justifications, the precise specification of the class of circular expressions to be considered is actually quite short.

Definition. A circular expression is *restricted* if each nonleaf is labelled with ***app***, ***upd***, or an uninterpreted operator symbol, and the following conditions are satisfied:

1. The first son of a vertex labelled ***upd*** is either a leaf or a vertex labelled ***upd***.

2. The first son of a vertex labelled ***app*** is either a leaf or a vertex labelled ***upd***. Moreover, there may not be an edge from a vertex labelled with an uninterpreted function symbol to a vertex labelled ***upd***.
3. The second son of a vertex labelled ***app*** or ***upd*** is a leaf. □

A useful reduction, which I will call "indirection elimination", can now be introduced. McCarthy [mcc62] observes that the following equivalence must hold:

$$\textbf{\textit{app}}(\ \textbf{\textit{upd}}(f,z,a)\ ,\ x\ )\ \equiv\ \textbf{\textit{if}}\ x=z\ \textbf{\textit{then}}\ a\ \textbf{\textit{else}}\ \textbf{\textit{app}}(f,x)$$

In particular, if x and z are given names, then the equality $x=z$ can immediately be tested. The following definition is more complicated than the above equivalence because it takes circularity into account.

Definition of indirection elimination. Let u be a vertex labelled ***app*** in a restricted circular expression. Let $v_1, v_2, \cdots, v_n$ be a maximal sequence of distinct vertices such that v_1 is the first son of u, and v_{i+1} is the first son of v_i, $1 \le i < n$. Note from the definition of restricted circular expressions that v_n is either a leaf or has as its first son v_i, for some $1 \le i \le n$.

If there exists a least j, $1 \le j \le n$, such that the second sons of u and v_j are for the same name, then u has the same value as the third son of v_j, so all edges into u can be redirected into the third son of v_j; otherwise, if v_n is a leaf, then make v_n the first son of u; otherwise, v_n is recursively defined and there is no update for the second son of u, so u represents $\perp$, so redirect all edges into u to a special leaf labelled Ω. □

It is a simple observation that repeated application of indirection elimination to a restricted circular expression eliminates all nonleaf vertices that correspond to static environments. In practice, if any leaves representing static environments are left after indirection elimination then an error has probably occurred since static environments generally start out with no information.

PROPOSITION . *Let G be a restricted circular expression whose root is not labelled* ***upd***. *Indirection elimination and the elimination of unreachable vertices results in a restricted circular expression G′ that has no vertices labelled* ***upd***.

Proof. By definition, the only vertices with edges into a vertex labelled ***upd*** have label ***app***. A simple induction on the number of vertices labelled ***upd*** establishes that the proposition must be true. □

The correctness of indirection elimination is left as an exercise for the reader.

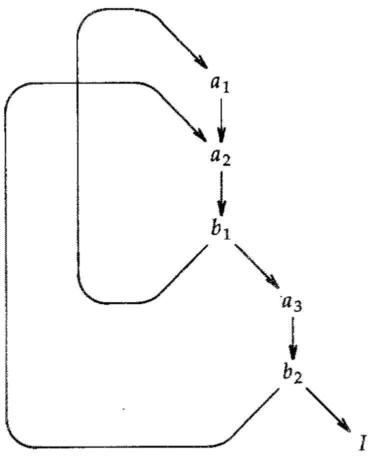

Figure 3. The flow diagram for a program in the text.

4. Application to label environments

The utility of circular expressions in a compiler generator based on denotational semantics will be illustrated by considering the following program fragment.

```
L1: u := v;
L2: w := x;
    if (p) goto L1;
    y := z;
    if (q) goto L2;
```

I have deliberately chosen a program with **goto** statements since structured constructs like **while** statements are actually easier to handle. In explaining this work, I have found it convenient to talk of flow diagrams. Given the above program, the flow diagram in Figure 3 might be drawn.

Consider however the problem of constructing a flow diagram *generator* that takes a program as input and constructs its flow diagram. One possibility is for the generator to make two passes over the candidate program. In the first pass, a table containing labels and their program points is constructed. Call this table an *environment*. In the second pass the environment is used to determine where a **goto** jumps to.

An environment can be thought of as a function, which when applied to a label yields the corresponding program point. If after the first pass the environment e' is determined, then Figure 4 shows an intermediate stage in the construction of the flow diagram.

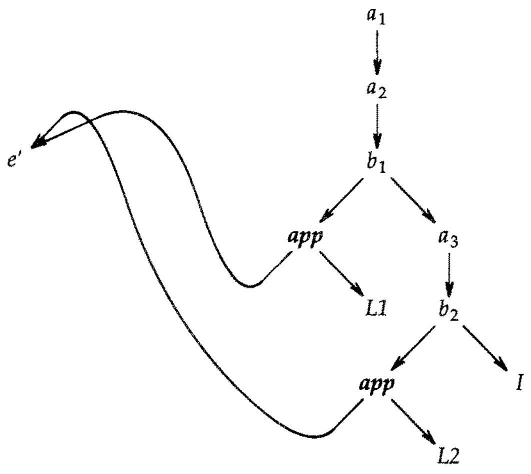

Figure 4. Once the environment e' containing the continuations for labels is determined, then the above circular expression gives the continuation for the whole program. Each of the operators is an abbreviation of an expression that maps continuations to continuations.

The flow diagram generator analogy is very close to the way denotational semantics of a language with **goto** statements are specified. In denotational semantics, instead of program points, "continuations" are associated with labels by the environment. *States*, which map identifiers to values, are not mentioned explicitly in a flow diagram. It is implicit that the final state on leaving a flow diagram depends on the initial state on entry to the diagram. The entry point is important — the mapping from initial state to final state depends on it. With an entry point L_0 of a diagram associate a *continuation* c_0, which is a function from the initial state to the final state. If the entry point is changed to L_1 from L_0 then there will be a different continuation c_1 associated with the entry point L_1.

The meaning of a label L is given by a continuation c that corresponds to making L the entry point of the program. Let I stand for the identity continuation.

The root of Figure 4 gives the continuation for the entire program. Since the first statement is labelled *L1* the root also gives the continuation for label *L1*. Irrelevant detail is avoided by sketching the effect of the assignments using the operators a_1, a_2, and a_3. These operators map the continuation following the relevant assignment to the continuation that includes the effect of the assignment.[1] Conditionals have two possible exits and there is a continuation associated with each exit. The operators b_1 and b_2, for the predicates p and q, take two continuations as arguments — the first corresponding to the predicate being true, the second to the predicate being false. A **goto** statement is handled by using the continuation of the label gone to: recall that the continuation for a label can be determined by applying a suitable environment to the label.

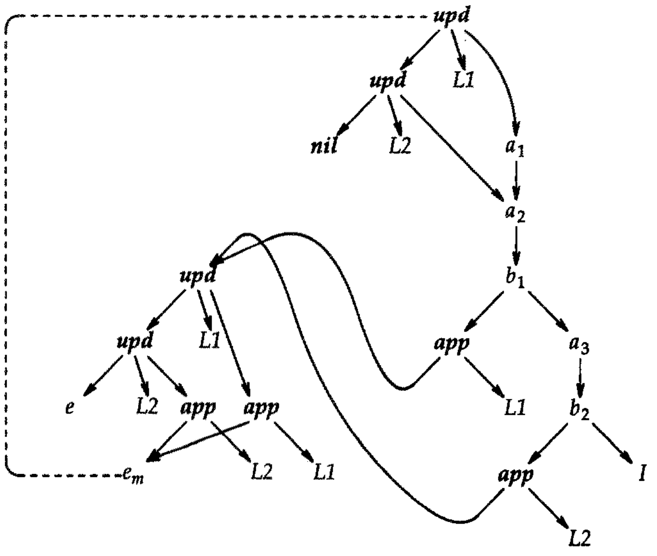

Figure 5. The dashed line connects vertices that are merged when an instance of the least fixed point operator is eliminated, as in Figure 2.

The suitable environment is determined as shown in Figure 5. There are two environments in Figure 5. Environment e_m is a *mini-environment* that contains label bindings for the current block alone. Ignoring the dashed line, let e_m be the initial environment that maps every label to the "undefined" continuation $\perp$. The environment e contains global bindings, so before it can be used, the continuations for the labels in the current block are entered into e yielding say e'. As in Figure 4, e' is used to determine the continuations for the whole program. An updated mini-environment is then determined by entering the continuations for the labels into an empty mini-environment ***nil***. Once ***fix*** is applied, Figure 5 is obtained.

As mentioned in Section 3, the following equivalence [mcc62] leads to a reduction called *indirection elimination.*

$$\textit{app}(\ \textit{upd}(f,x,a)\ ,\ y\ )\ \equiv\ \textit{if}\ y{=}x\ \textit{then}\ a\ \textit{else}\ f(y)$$

In the graph in Figure 5, the labels *L1* and *L2* are literal labels, not label variables, so the equality $y{=}x$ can immediately be tested. Look at the application of e_m to *L1* at the bottom left hand corner in Figure 5. Since the vertex for e_m is merged with the vertex connected to it with the dashed line, there is an application of an update as in the equation above. In Figure 6, rather than going

[1] The precise specification of a_1, the operator for the first assignment, is as follows. Given continuation c,

$$a_1(c)\ =\ \lambda s.\ c(\ s[\ s[\![v]\!]\ /\ u\ ]\ )$$

The remaining operators in Figure 4 are also continuation transformers.

through the mini-environment, there is a direct edge to the continuation for *L1*. Similar remarks apply to the label *L2*. Vertices that are unreachable have been eliminated.

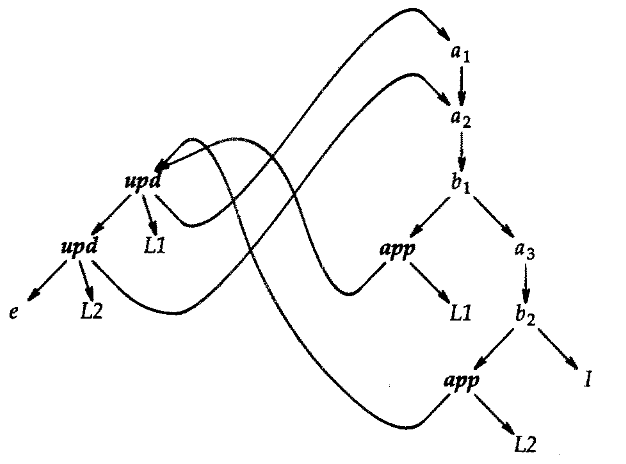

Figure 6. This graph is obtained from the graph in Figure 5 by eliminating the indirection through the mini-environment.

Indirection elimination can be applied to Figure 6 as well. This time, elimination of the ***app*** and other vertices that become unreachable yields the circular expression in Figure 3! Only now the operators in Figure 3 have a very definite meaning as continuation transformers.

5. A compiler generator

Figure 7 shows the logical organization of the front end of a semantics directed compiler generator. The user interface is indicated by the dotted box. For a candidate language *L*, a denotational semantics *Lsem.d* and regular expressions *Ltok.r* specifying the lexical structure of *L* have to be provided.

Together with a package of trivial symbol table routines, the regular expressions in *Ltok.r* constitute *Ltok.l*. The program Lex [les75] generates a lexical analyzer for *L* from *Ltok.l*.

The program *d2y* examines the semantic rules in *Lsem.d* and converts them into a C [ker78] program fragment that will cause a graph to be built for a program in *L*. The mapping performed by *d2y* is such that its output *Lsem.y* can be piped through the parser generator Yacc [joh75] to construct a syntax analyzer for programs in *L*.

Given a program *prog.L* in *L*, the output of the syntax analyzer is a graph representing the meaning of the program. The *reducer* performs β-reductions, indirection elimination, and the selection of elements from lists. (Other reductions are easy to add.) β-reduction is performed only when the ***app*** vertex is the sole father of the λ vertex. (Common subexpression elimination may be needed to permit some β-reductions. Presently, this has been done by coding *Lsem.d* carefully. Eventually the algorithm of [dow80] will be used.) The output of the reducer is a simplified circular expression.

Figure 7 also shows how the program *d2y* is constructed. The notation used for writing denotational specifications like *Lsem.d* is inspired by that of Yacc. The major difference is that instead of a C program fragment, the semantic rules are denotational. Further details are given in Appendix B. *metaspec.l* and *metaspec.y* give the syntax of the notation in which *Lsem.d* is written.

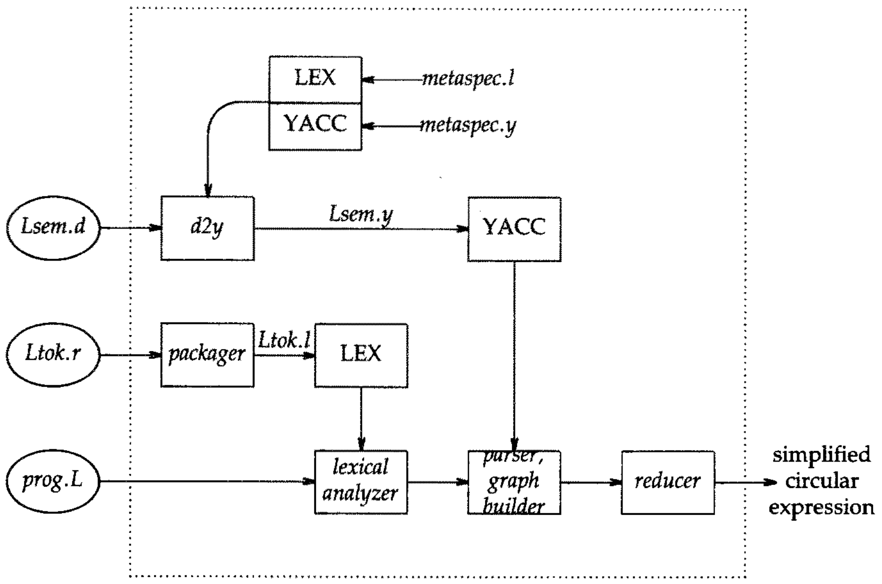

Figure 7. Logical organization of the front end of a semantics directed compiler generator *d2*. For a language *L*, the lexical and syntactic analyzers are constructed from the specifications *Ltok.r* and *Lsem.d*, respectively. *Ltok.r* contains regular expressions and their translations, while *Lsem.d* contains denotational semantic rules.

6. Discussion

The few examples that I have tried are very encouraging. The present version of *d2y* has 31,876 bytes (on a PDP 11/70: November 9, 1980). In each of the examples, the lexical analyzer, parser, and reducer together are less than 27,500 bytes. (These programs contain fat that can easily be trimmed. By way of a lower bound, empty Lex and Yacc specifications lead to a lexical analyzer and parser containing 9,288 bytes.) Besides the **goto** language of Section 4 and Appendix B, I have tried a **while** language with **break** and **continue** statements; the language LC, a version of the lambda calculus from [mos79b]; and the Loop language of [ten76].

Based on a meagre five samples, the reducer tends to eliminate at least two thirds of the vertices constructed by the syntax analyzer. This figure is not surprising. For example, the meanings of statements are typically functions from environments to continuations to continuations. In the graph built by the syntax analyzer there will be λ vertices and bound variable vertices for environments and continuations. Often the λ vertices have one father, which is an ***app*** vertex, so β-reduction is possible. Each β-reduction eliminates three vertices. For the example program in Section 4, the final circular expression is actually smaller than Figure 3 suggests since β-reduction will eliminate the links between some of the continuation transformers. The actual output is shown in Appendix B.

Circular expressions are not an end in themselves. They can either be interpreted using reductions like the ones listed by Mosses [mos79a] (see also [hof79, hue79]), or code can be generated directly from them. I am presently examining ways of converting applicative expressions (with cycles) to imperative code, but it is too early to talk of progress on this problem. In the meantime, techniques that require additional information for each language can be used. For example, Wand [wan80] defines the semantics of languages using combinators that are easy to implement directly on a machine. Using such combinators the simplified circular expression constructed as in Figure 7 will be easy to map onto machine code. The approach of [ras80] uses detailed knowledge of continuations, but can be applied to the reduced circular expression. In short, all the approaches to compiler generation that I am aware of would benefit by starting with the reduced circular expression.

Acknowledgements

I would like to thank Gerard Berry for helpful discussions, Doug McIlroy for providing some perspective on early work on semantics, Bjarne Stroustrup for discussions on programming using classes [sts80], and Al Aho for comments on the presentation. Lex by Mike Lesk and Yacc by Steve Johnson were invaluable tools. The figures were drawn using Brian Kernighan's Pic program [ker80].

Appendix A. Semantics of circular expressions

A *circular expression* is a directed graph with a unique vertex called a *root*. The meaning of a circular expression is given by starting at the root and unfolding the graph into an infinite tree. The purpose of this section is to suggest how the semantics of circular expressions can be formalized.

The well known idea of cutting loops and associating recursive equations with a flow diagram [mcc62] can be applied just as easily to circular expressions. (The system of recursive equations that is generated will be referred to as a *recursive scheme*.) The semantics of a circular expression can then be given by the semantics of a recursive scheme for it. See [sco71, cou76, adj77, rey77] for example. The basic idea in each of these papers is to construct an infinite expression tree and let the meaning of the tree be the meaning of the recursive scheme. Unfolding the circular expression yields the same tree.

The semantics of an infinite expression cannot be given by associating values with the leaves and working up to the value of the root, since the root can never be reached. Its meaning is therefore given as the limit of an infinite sequence of values of finite expressions. For limits to exist, the expressions in the sequence and their values must satisfy certain constraints.

On a syntactic level, each expression tree in the sequence represents a stage in the incremental unfolding of a circular expression (graph). There will be leaves in these expression trees at which further unfolding needs to take place, and this fact is noted by labelling such leaves with the symbol Ω. Along with Ω comes a partial order on expressions, since Ω, representing the fully folded expression, gives less information than any other expression. (More precisely, no more information than any other expression.) For limits to exist in general, infinite expressions have to be added. There is a similar partial order on values.

Since functions may be defined and applied in circular expressions, some of the work on the λ-calculus is also quite relevant. Following Wadsworth, a special symbol Ω, whose value is $\perp$, is added to the λ-calculus in [wad78, hyl76, lev76, wel75]. A notion of "approximant" of a term is defined (this is more than just direct approximation since β-reductions can take place). The value of a term is shown to be the limit of the values of all its approximants [wad78, hyl76]. Hyland shows the result for both the $\mathbf{D}_\infty$ [sco73] and the $\mathbf{P}\omega$ [sco76] models. Such results allow equivalences to be proved between a finite expression on the one hand and an infinite expression on the other.

Appendix B

The notation used to specify semantics will be illustrated by considering a simple language with **goto** statements. The specification consists of three parts: declarations that help Yacc construct a parser; domain declarations; and, the syntactic and semantic rules. The three parts are delimited using the marker `%%`.

In the first part, *d2y* looks at lines beginning with `%token` and `%start`. All other declarations are copied through for the benefit of Yacc. Knowledge of tokens is used when the semantic rules are examined. The start symbol is captured and a new start symbol supplied so that some initializing code can be executed before parsing of a program begins. In particular, if there are any predeclared identifiers or hidden identifiers in the semantics, they have to be passed through and entered into the symbol table so that their special status is recognized when a program is parsed.

At present the domain declarations are examined, but the type information is not used to do any checking. As an aside, domain declarations are represented by a directed graph, where cycles record reflexive domain declarations. Associated with each domain identifier is a pointer to a vertex in the directed graph. Along with the domains are: `var` declarations of variables that represent elements of the domain; `nont` declarations that identify nonterminals in the syntax whose meanings belong to the domain; and, `elem` declarations that identify elements of the domain. So far, I have used `elem` declarations for hidden identifiers in a language with **break** and **continue** statements, and for the identifiers `plus` and `mult` that have predefined meanings in LC [mos79b].

The third part contains the syntactic and semantic rules. If all the text between braces is deleted, a syntactic specification in the input language of Yacc will be left. The semantic rules between braces are in a metalanguage similar to DSL [mos79a] without the parse tree constructs. The precedence of operators is different from that in DSL. For those unfamiliar with DSL, the metalanguage is an extended λ-calculus.

In the following specification, syntactic rules are indented one tab stop, while semantic rules are indented two tab stops. The meaning of a syntactic object is indicated by preceding it with a `$` sign. For ease of implementation, the specification of all valuations is collected together, so the meaning of a syntactic object will in general be a function from some domain to a list of elements from some other domains.

There are differences of detail between the discussion in Section 4 and the exact treatment of labels in the semantic specification that follows. For pedagogic reasons, the mini-environment was constructed using functions rather than lists. Here, the meaning of a statement is a function from environments and continuations to a list containing a continuation as a second element and with label information as a first element. The label information consists of a list of pairs of labels and their associated continuations. There is an operation in the metalanguage that allows a function to be be updated using a list, so that the list of labels and their associated continuations can be used to directly update the environment.

The code for *Lsem.d* follows.

```
%token GOTO IDE IF
%start prog
%%
     Ide                           var IDE;
     V;
     S = Ide -> V                  var s;
     A;
     C = S -> A                    var c;
     Env = Ide -> C                var e;

     S -> V                        nont exp;
     Env -> C -> C                 nont prog;
     Env -> C -> [ [Ide,C]* , C ]nont stm, stmlist;
%%
prog  : stm
           { lambda e. lambda c.
                let  cyclic p = $stm (e[p.1]) c;
                in   p.2
           }
      ;
```

`/*` The keyword `cyclic` indicates that the definition of `p` is circular. The environment `e` is updated using the list `p.1`, i.e. the first element of the pair `p`. `p.1` is itself a list of pairs of labels and associated continuations. The second element `p.2` of `p` is the continuation that is associated with the beginning of the program.
`*/`

```
stm   : IDE ':' '=' exp ';'
           { lambda e. lambda c.
             let sc = lambda s. c(s[$IDE;$exp s]);
             in  ((),sc)
           }
      | '{' stmlist '}'
```

```
            { $stmlist }
        | IDE ':' stm
            { lambda e. lambda c.
              let pair = $stm e c;
                  sc = pair.2;
                  sl = ($IDE,sc) cons pair.1;
              in (sl,sc)
            }
        | IF '(' exp ')' GOTO IDE ';'
            { lambda e. lambda c.
              let sc = lambda s.
                         $exp s -> e ($IDE) s , c (s);
              in ((),sc)
            }
        ;
exp     : IDE
            { lambda s. s($IDE) }
        ;
stmlist     : stm
            { $stm }
        | stmlist stm
            { lambda e. lambda c.
              let p2 = $stm e c;
                  sc = p2.2;
                  p1 = $stmlist e sc;
                  sl = p1.1 cat p2.1;
              in (sl,p1.2)
            }
        ;
```

The mapping performed by *d2y* is such that the first semantic rule will be transformed to:

```
prog: stm
          {
            nodeptr[3] = ctnode(VAR,9,12);
            nodeptr[4] = ctnode(VAR,7,10);
            nodeptr[5] = ctnode(VAR,99,16);
            nodeptr[6] = $1;
            nodeptr[7] = ctnode(SELECT,1,nodeptr[5]);
            nodeptr[8] = ctnode(UPD,nodeptr[3],nodeptr[7]);
            nodeptr[9] = ctnode(APP,nodeptr[6],nodeptr[8]);
            nodeptr[10] = ctnode(APP,nodeptr[9],nodeptr[4]);
            cttie(nodeptr[5],nodeptr[10]);
            nodeptr[11] = ctnode(SELECT,2,nodeptr[10]);
            nodeptr[12] = ctnode(LAM,nodeptr[4],nodeptr[11]);
            nodeptr[13] = ctnode(LAM,nodeptr[3],nodeptr[12]);
            $$ = nodeptr[13];
          }
      ;
```

Until a program is supplied for parsing, the "address" of the vertices constructed for the program are not available. However, when `ctnode` creates a vertex during parsing, it can leave the "address" in the array `nodeptr` where subsequent calls to `ctnode` will be able to find it. The routine `cttie` creates the circular expression.

Once the parser, graph builder, and reducer have been constructed, the following program will be mapped to the circular expression below.

```
{
```

```
L1:   u := v;
L2:   w := x;
      if(p) goto L1;
      y := z;
      if(q) goto L2;
}
```

In the representation of the circular expression below, vertex numbers appear before the colon on the left. When a number appears by itself on the right of a colon, it represents the expression rooted at the vertex with that number. Variables like `s10` and `s41` have been created to distinguish bound instances of the same variable `s`.

```
  6: s10 ( v )
 14: s10 [ 6 / u ]
 15: 47 ( 14 )
 16: lam s10 . 15
 37: s41 ( x )
 45: s41 [ 37 / w ]
 47: lam s41 . 94
 83: 45 ( p )
 91: 16 ( 45 )
 94: 83 -> 91 , 160
117: 45 ( z )
125: 45 [ 117 / y ]
149: 125 ( q )
157: 47 ( 125 )
158: c182 ( 125 )
160: 149 -> 157 , 158
188: lam c182 . 16
189: lam e181 . 188

                  root  189
```

There is no mystery to the routine used for printing out circular expressions. There is a sequential scan through the vertices during which vertices for identifiers, variables, and constants are ignored. At each vertex representing an operator some printing takes place. The details can be gleaned from the example.

The astute reader will have noticed that at least 189 vertices were constructed for the above program, of which about a sixth remain in the reduced circular expression. I suspect that the phases of graph building and reduction can be made into coroutines without too much difficulty, so that it will not be necessary to construct large intermediate expressions that get cut down to size.

References

adj77. adj: J. A. Goguen, J. W. Thatcher, E. G. Wagner, and J. B. Wright, "Initial algebra semantics and continuous algebras," *J. ACM* **24**(1), pp. 68-95 (January 1977).

aho79. A. V. Aho and J. D. Ullman, "Universality of data retrieval languages," *Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio TX, pp. 110-120 (January 1979).

chu41. A. Church, *The calculi of lambda conversion*, Annals of Math. Studies, No. 6, Princeton University Press, Princeton NJ (1941).

cou76. B. Courcelle and M. Nivat, "Algebraic families of interpretations," *17th Annual Symposium on Foundations of Computer Science*, Houston TX, pp. 137-146 (October 1976).

dow80. P. J. Downey, R. Sethi, and R. E. Tarjan, "Variations on the common subexpression problem," *J. ACM*, pp. 758-771 (October 1980).

gor79. M. J. C. Gordon, *The Denotational Description of Programming Languages*, Springer-Verlag, New York NY (1979).

hof79. C. M. Hoffman and M. J. O'Donnell, "An interpreter generator using tree pattern matching," pp. 169-179 in *Sixth Annual ACM Symposium on Principles of Programming Languages*, San Antonio TX (January 1979).

hue79. G. Huet and J. J. Levy, "Call by need computations in non-ambiguous linear term rewriting systems," Rapport Laboria 359, IRIA (August 1979).

hyl76. M. Hyland, "A syntactic characterization of the equality in some models for the lambda calculus," *J. London Math Soc, Second Series* **12**(3), pp. 361-370 (February 1976).

joh75. S. C. Johnson, "Yacc — yet another compiler compiler," CSTR 32, Bell Laboratories, Murray Hill NJ (July 1975). See the *UNIX Programmer's Manual* **2** Section 19 (January 1979)

jon80. N. D. Jones (ed), in *Semantics-Directed Compiler Generation*, Lecture Notes in Computer Science 14, Springer-Verlag, Berlin (1980).

ker78. B. W. Kernighan and D. M. Ritchie, *The C Programming Language*, Prentice-Hall, Englewood Cliffs NJ (1978).

ker80. B. W. Kernighan, "PIC - A crude graphics language for typesetting," CSTR, Bell Laboratories, Murray Hill NJ (1980).

lan64. P. J. Landin, "The mechanical evaluation of expressions," *Computer J.* **6**(4), pp. 308-320 (January 1964).

les75. M. E. Lesk, "Lex — a lexical analyzer generator," CSTR 39, Bell Laboratories, Murray Hill NJ (October 1975). See the version by M. E. Lesk and E. Schmidt in the *UNIX Programmer's Manual* **2** Section 20 (January 1979)

lev76. J.-J. Levy, "An algebraic interpretation of the λβκ-calculus and an application of labelled λ-calculus," *Theoretical Computer Science* **2**(1), pp. 97-114 (June 1976).

mcc62. J. McCarthy, "Towards a mathematical science of computation," pp. 21-28 in *Information Processing 1962*, ed. C. M. Popplewell, North-Holland, Amsterdam (1963).

mil76. R. E. Milne and C. Strachey, *A Theory of Programming Language Semantics, 2 Vols.*, Chapman and Hall, London, and John Wiley, New York (1976).

mos79a. P. D. Mosses, "SIS — semantics implementation system: Reference manual and user guide," DAIMI MD-30, Department of Computer Science, University of Aarhus, Denmark (August 1979).

mos79b. P. D. Mosses, "SIS — semantics implementation system: Tested examples," DAIMI MD-33, Department of Computer Science, University of Aarhus, Denmark (August 1979).

pac74. G. Pacini, C. Montangero, and F. Turini, "Graph representation and computation rules for typeless recursive languages," pp. 158-169 in *Automata, Languages and Programming, 2nd Colloquium, Saarbrucken*, Lecture Notes in Computer Science 14, Springer-Verlag, Berlin (1974).

par70. D. M. R. Park, "The Y combinator in Scott's lambda-calculus models," unpublished, University of Warwick (1970).

ras80. M. Raskovsky, "Compiler generation and denotational semantics," *see* [jon80].

rey77. J. C. Reynolds, "Semantics of the domain of flow diagrams," *J. ACM* **24**(3), pp. 484-503 (July 1977).

sco71. D. S. Scott, "The lattice of flow diagrams," pp. 311-372 in *Symposium on Semantics of Algorithmic Languages*, ed. E. Engeler, Lecture Notes in Mathematics 188, Springer-Verlag, Berlin (1971).

sco73. D. S. Scott, "Lattice theoretic models for various type-free calculi," pp. 157-187 in *Proc. IVth International Congress for Logic, Methodology and the Philosophy of Science, Bucharest*, ed. P. Suppes *et al.*, North-Holland, Amsterdam (1973).

sco76. D. S. Scott, "Data types as lattices," *SIAM J. Computing* **5**(3), pp. 522-587 (September 1976).

sta79. J. Staples, "A graph-like lambda calculus for which leftmost-outermost reduction is optimal," pp. 440-455 in *Graph-Grammars and Their Application to Computer Science and Biology*, Lecture Notes in Computer Science 73, Springer-Verlag, Berlin (1979).

sto77. J. E. Stoy, *Denotational Semantics*, MIT Press, Cambridge MA (1977).

str72. C. Strachey, "Varieties of programming language," pp. 222-233 in *International Computing Symposium*, Cini Foundation, Venice (April 1972).

sts80. B. Stroustrup, "Classes: An abstract data type facility for the C language," CSTR 84, Bell Laboratories, Murray Hill NJ (April 1980).

ten76. R. D. Tennent, "The denotational semantics of programming languages," *Comm. ACM* **19**(8), pp. 437-453 (August 1976).

wad76. C. P. Wadsworth, "The relation between computational and denotational properties for Scott's D_∞-models of the lambda calculus," *SIAM J. Computing* 5(3), pp. 488-521 (September 1976).

wad78. C. P. Wadsworth, "Approximate reductions and lambda calculus models," *SIAM J. Computing* 7(3), pp. 337-356 (August 1978).

wan80. M. Wand, "Deriving target code as a representation of continuation semantics," TR 94, Computer Science Department, Indiana University, Bloomington IN (July 1980).

wel75. P. H. Welch, "Continuous semantics and inside-out reductions," pp. 122-146 in *λ-calculus and Computer Science Theory*, Lecture Notes in Computer Science 37, Springer-Verlag, Berlin (1975).