Generating Execution Facilities for Integrated Programming Environments

by

Anil Allan Pal

Computer Sciences Technical Report #676

November 1986

Generating Execution Facilities

for

Integrated Programming Environments

by

Anil Allan Pal

A thesis submitted in partial fulfillment of the
requirements for the degree of

Doctor of Philosophy
(Computer Sciences)

at the

UNIVERSITY OF WISCONSIN - MADISON

1986

Abstract

This thesis presents an approach to the problem of generating execution facilities for integrated programming environments from specifications of the dynamic semantics of programming languages. The approach is based on techniques used in semantics-directed compiler generators, using a denotational semantic description of the language. These techniques are adapted to the special nature of an integrated programming environment, in particular the need to provide *incremental translation* and *interactive execution.*

In interpreters generated using our system, programs are translated into denotations that are represented as linked structures containing pointers to the compiled code of denotation functions. This representation is compact, provides reasonable execution efficiency, and is easy to maintain incrementally during program modification.

The correspondence between the executable representation and the parse tree of the program can be exploited to permit the user to interact with the program at runtime in terms of source-language constructs, thus providing support for interactive execution. We show how many of the features of previous hand-coded integrated programming environments can be incorporated naturally into the generated execution facilities.

Acknowledgements

Many people have, directly and indirectly, contributed to the completion of this thesis. First thanks must go to my advisor, Charles Fischer, who has been my guide and inspiration from the time I started graduate school to the present. Thomas Reps provided many constructive comments on my ideas and their expression in the thesis; I also owe him thanks for the use of the Synthesizer Generator. Thanks also to Susan Horwitz, who carefully read several drafts of the thesis and made many suggestions that greatly improved the final version.

Among my fellow graduate students, Prasun Dewan deserves special mention for being a constant companion throughout my stay in Madison; as both a friend and colleague, he kept me going through the best of times and the worst of times. Thanks to Michael Scott for the use of GFE, to Kishore Ramachandran and Toby Lehman for the thesis formatting macros, to Anoop Gupta for providing many interesting papers and technical reports, and to Bill Kalsow for, among other things too numerous to list, maintaining the Modula-2 compiler. Thanks also to other members, past and present, of my research group, including Jon Mauney, Greg Johnson, Dan Stock, Felix Wu, Will Winsborough, and G. Venkatesh, for valuable ideas and helpful discussions.

I would also like to thank the Department of Computer Science and the National Science Foundation for their financial support. Thanks also to the secretarial staff of the department, especially Joyce, Laura, Lorene, Marilyn, Pat, Sharon, Sherry and Sheryl, for all their help.

On a more personal level, my parents, Ajoy and Kathleen Pal, have always encouraged and supported me in all I have done; I owe them a debt greater than I can express or ever hope to repay. My brother Allan, together with Monica, provided invaluable help during the final rush to finish. I would also like to thank all the people associated with "The Pitz," including Viranjit, Anoop, Sriram, Chou, Mohan, Usha, Kapoor, Patti, Cathy, and Greta, for providing a haven away from the pressures of academics. Thanks also to David Aslakson, Allan Bricker, David Carwardine, Karen Fortney, Tom Grandine, Dave Holland, Nuzhat Khan, Karl Knapp, Janet Lumsden, and Arvind and Geeta Rao for their friendship and company at various times during my stay in Madison. Special thanks go to Kathy Smoot for being an understanding friend during the last year.

Finally, I would like to express my gratitude to my son, Imran, who has been a source of inspiration, comfort, and joy since the time he was born.

Table of Contents

List of Figures

Chapter 1

Introduction

The coding phase of program development usually involves many iterations of an *edit-compile-debug* cycle. During this phase the programmer, using a variety of tools, successively modifies, translates, and tests a program until it appears to be correct. The set of tools available to the programmer during this process greatly affects the time taken to complete it. Usually, the tools used are an editor (to write and modify programs), a compiler (to translate them), and a debugger (to help deduce what went wrong).

One approach to improving the efficiency of the coding process is to combine these tools into a single *integrated* tool, often called an *environment*. For example, several editors use the knowledge that the domain of objects being edited is limited to programs in a particular programming language to assist the programmer in the creation of "correct" programs. These editors, variously known as *language-based editors*, *syntax-directed editors*, or *structure editors*, provide assistance akin to that provided by the front end of a traditional compiler; syntactic and static semantic errors may be detected and perhaps repaired automatically. In some cases, program entry and modification are not textual, but rather in terms of the syntactic units of the language, thus assuring syntactic correctness at all times.[1]

---

[1] Programs may, however, still be *incomplete*.

The next step is to add to such an editor the capability to *run* programs, thus creating a complete integrated programming environment, or IPE. Since we are concerned with program *development*, an IPE should provide adequate debugging support during program execution. In addition, it is desirable that the execution facilities exploit the special nature of an integrated environment (as opposed to a traditional editor and compiler) to shorten the development cycle as much as possible. Allowing rapid transitions from editing to execution, and vice versa, helps achieve this goal.

In view of the large number of programming languages in existence, there is a strong incentive to automate the creation of IPEs, rather than writing each one from scratch. In order to develop an IPE generator, we need a language-independent skeleton or kernel, together with formal specification of all the language-dependent aspects of an IPE, and algorithms for deriving efficient implementations from those specifications. For a minimal complete IPE, the language-dependent aspects are the *syntax*, *static semantics*, and *dynamic semantics* of the language.

This thesis presents an approach to generating execution facilities for IPEs from specifications of the dynamic semantics of programming languages. The approach is based on techniques used in semantics-directed compiler generators, using a denotational semantic description. These techniques are adapted to the special nature of an IPE, in particular the need to provide *incremental translation* and *interactive execution*.

The remainder of this introduction is arranged as follows: First, section 1.1 outlines the separation of the various components of a language description. Next, section 1.2 describes briefly previous approaches to the generation problem. Finally, section 1.3 introduces our method.

1.1. Syntax and Semantics

When describing languages, we try to separate questions of form from questions of meaning; the former we term syntax, and the latter, semantics. [Tennent81] provides a good starting point for the reader interested in further discussions of the various aspects of programming language specification.

1.1.1. Syntax

The syntax of a programming language specifies the form of programs in the language. Syntax descriptions are usually written in a form of context-free grammar or BNF (Backus-Naur Form), first developed to describe the syntax of the programming language ALGOL 60 [Naur63]. This notation gives a precise, yet easy to understand, specification of a language's syntax. Additionally, by restricting our descriptions to certain classes of grammars, we can automatically construct efficient parsers that determine the syntactic structure of programs.

It is often useful to distinguish between the *abstract syntax* and *concrete syntax* of a language. The concrete syntax gives the actual form of valid programs as they would be printed out or typed in, including all the "syntactic sugar" that is necessary for parsing, whereas the abstract syntax discards all superficial elements, leaving only the part essential to the meaning of the program. For example, Figure 1-1 gives the abstract and concrete syntax of an **if** statement in Pascal [Jensen78]. Descriptions of languages routinely include a formal concrete syntax specification in a form similar to that in Figure 1-1, and it is a fairly straightforward matter to write such a specification for a conventional programming language.

Concrete Syntax

<statement> ::= **if** <condition> **then** <statement> <else part>
<else part> ::= **else** <statement>
::=

Abstract Syntax

<statement> ::= **if-then** (<condition> <statement>)
::= **if-then-else** (<condition> <statement> <statement>)

Figure 1-1: Concrete and Abstract Syntax of an if statement.

1.1.2. Semantics

Where syntax specifies the form of valid programs in a language, semantics ascribes meaning to them. There are several aims in specifying the semantics of programming languages. A language semantics may be purely descriptive in intent, or it may be designed to aid in some other goal, such as implementing a language processor, or proving certain properties of programs written in the language.

While there are standard, well-understood formal methods for specifying the syntax of programming languages, the same cannot be said for semantics. All too often, the only semantic specification of a language is a document in a natural language such as English, with all the ambiguity and room for error that entails. There are, however, formal semantic specification techniques that have been developed.

The tractability of programming language syntax (in contrast to semantics) makes it the ideal base from which to launch sorties against the formidable tasks of language specification and implementation. Semantic descriptions that have a structure guided by the syntax of the language are referred to as *syntax-directed*. Because the abstract

syntax is stripped of information that is irrelevant to the meaning of the program, it is often chosen as the basis for syntax-directed specification of the semantics of a language.

The semantics of languages can be divided into *static* and *dynamic* components. We include in static semantics all those aspects of a language that are usually checked by a compiler, such as type correctness and variable declarations. Since these checks can also be considered to be further constraints on the form of correct programs, they are sometimes called *context-sensitive syntax*, as opposed to what we have termed syntax, which is *context-free*. The dynamic semantics of a language specifies the run-time behavior of programs. The simplest way to describe the purview of dynamic semantics is by default; dynamic semantics includes everything that is not covered by syntax or static semantics.

The next two sub-sections present a short introduction to attribute grammars and denotational semantics, respectively. Attribute grammars have been used as the basis of language-based editor generator systems (the precursors of integrated programming environments), and denotational semantics form the basis of our method for IPE generation.

1.1.2.1. Attribute Grammars

Attribute grammars were proposed by Knuth [Knuth68] in order to extend context-free grammars to include context-sensitive properties of programming languages. A fixed set of *attributes* is associated with each grammar symbol. These attributes represent information associated with the symbol, such as its type or value.

Each production in the syntax of the language has associated with it a set of attribute evaluation rules. If an attribute rule associated with a production defines the value

of an attribute of the symbol on the left-hand side of the production, we refer to that attribute as a *synthesized* attribute; if the rule defines the value of an attribute of a symbol on the right-hand side of the production, we refer to the attribute as an *inherited* attribute. The attribute rules may only use attribute values associated with symbols within the production in question. Figure 1-2 presents a simple example of type checking in an expression. Note that <expression>$_n$ refers to the n^{th} occurrence of *<expression>* in the production, and *<expression>*.type refers to attribute *type* of node *<expression>*.

Apart from the constraint on the attributes evaluation rules can use, there is no restriction on the functions that may be specified. These functions can be arbitrarily complex, and indeed may not even be defined for all inputs. Attribute grammars are thus extremely powerful, and can in theory be used to specify all aspects of the semantics of programming languages, although their strength lies in describing static semantics.

Production:

<expression> ::= *<expression>* + *<expression>*

Attribute Rule:

<expression>$_1$.type = **if** <expression>$_2$.type == <expression>$_3$.type
then <expression>$_2$.type
else *error-type*.

Figure 1-2: An attribute grammar example.

1.1.2.2. Denotational Semantics

Denotational semantics, as pioneered by Dana Scott and Christopher Strachey [Scott71], describes the semantics of programming languages by mapping programs in the language into mathematical functions. This mapping is *denotational*, in that the meaning of a composite construct (such as an **if** statement) is defined in terms of the meanings of its immediate constituents (an expression and another statement). Chapter 2 gives a more detailed introduction to denotational semantics; we present here some of the salient features of this semantic description method.

A denotational description of a programming language is syntax-directed and modular. Every syntactic construct in the language is mapped into a mathematical object (such as a number or function) that models its meaning. The meanings of elementary constructs (such as numerals or identifiers) are specified directly, whereas the meanings of composite constructs are defined in terms of the meanings of their immediate constituents.

In addition to the abstract syntax of the language, a denotational semantics must also specify the *semantic domains* into which language constructs will be mapped, as well as the *semantic functions* that define the mapping. There is usually one semantic function for each syntactic category in the language, defined by cases on the productions in the abstract syntax.

As an example, Figure 1-3 gives the semantics of binary numerals (sequences of 0s and 1s terminated by a "B"), following Tennent [Tennent81]. The semantic domain is **N**, the non-negative integers. The semantic function, named **E**, maps binary numerals (members of **BinInt**) into non-negative integers (members of **N**). To differentiate between syntactic and semantic objects which have similar representations, such as the binary digit "0" and the integer zero, we will enclose syntactic objects in

brackets ([and]) in the definitions of semantic functions. The operators used in the semantic function ("*", "+", "=") are the standard integer operators.

Denotational semantics can be used to specify the entire (static and dynamic) semantics of programming languages such as Pascal [Tennent78]. The denotational nature of the definition permits it to be compact and modular. The use of mathematical entities as the medium of specification facilitates manipulations for proving properties of programs, languages, and implementations. For our purposes, however, probably the most important characteristic of denotational definitions is that they can be used to generate implementations of the language.

Abstract Syntax

BN$\in$ **BinInt**

BN $\rightarrow$ Seq B
Seq $\rightarrow$ 0 | 1 | Seq 0 | Seq 1

Semantic Domain

N = {0,1,2, ...}

Semantic Function

E: **BinInt** $\rightarrow$ **N**

E[0] = 0
E[1] = 1
E[Seq 0] = 2 $\times$ **E**[Seq]
E[Seq 1] = 2 $\times$ **E**[Seq] + 1

Figure 1-3: Denotational semantics of binary numerals.

1.2. The State of the Art

Previous approaches to the problem of IPE generation have used a variety of specification techniques. The ALOE system [Medina-Mora82] developed as part of the Gandalf [Habermann82] project at Carnegie-Mellon University provides an editor kernel and a generator that combines a syntax description with the kernel to produce a syntax-directed editor. The implementor can add semantic "action routines" and associate an action routine with a particular production in the syntax. The ALOE kernel will invoke the appropriate action routine whenever a node defined by a particular production is edited. The action routines are written in a conventional programming language and may perform any action that can be expressed in that language. The strength of this approach, its generality, is also its greatest weakness; since there are no constraints on what an action routine may do, it is hard to ensure that it does what is required, and nothing else. Later work on ALOE [Ambriola84] uses a special language for writing action routines and performs static and dynamic checks to ensure the syntactic integrity of program trees.

A more formal approach uses attribute grammars to specify the semantics of languages. Efficient algorithms have been developed for the *incremental* evaluation of attributes [Reps83, Johnson83] when a syntax tree changes during editing. Editor generators such as the Synthesizer Generator [Reps84] and Poegen [Fischer84] use these algorithms to provide automatically-generated editors that perform full syntactic and static-semantic checking incrementally during program editing. The declarative nature of attribute grammars makes specification easier than the action routine model, especially in the case of incremental changes, where order-dependent errors [1] are possible

[1] By order-dependent errors we mean situations where the final values of attributes depend not only on the current state of the program, but also on the sequence of editing operations performed to reach that state.

in an imperative specification, but not in an attribute grammar.

The PSG programming system generator developed at the Technical University of Darmstadt [Bahlke85] generates complete IPEs from formal definitions. The static semantics of the language are specified using the context condition formalism presented in [Snelting86] and the dynamic semantics are described denotationally using a functional language based on the lambda calculus.

Kaiser [Kaiser85] proposes the use of *action equations*, which are claimed to extend attribute grammars to specify the run-time semantics of languages. Kaiser's approach has not yet been incorporated into an IPE generator.

Ambriola and Montangero [Ambriola85] describe a generator of execution facilities for Gandalf environments that is based on a denotational semantic specification of the dynamic semantics of the language, although they do not perform incremental processing.

1.3. IPE generation based on Denotational Semantics

A truly integrated programming environment must provide facilities for execution of programs developed within the environment. In order to gain the maximum benefit from the use of such an environment, the execution facilities should be fully integrated into the overall environment, rather than being added as an afterthought, and should exploit this integration to provide facilities not easily supplied by traditional discrete environments.

The features we desire can be broadly described as *incremental translation* and *interactive execution*. We would like to reduce as much as possible the sometimes long compilation delay encountered in a traditional environment. Since we are concerned primarily with program *development*, we are willing to accept reduced execu-

tion efficiency in return for increased translation speed, in the hope that the overall length of an *edit-compile-debug* cycle will thereby decrease. Ideally, we would like to maintain an executable version of the program at all times during editing, thus creating the illusion of *instant* translation. The processing necessary to update the executable version in response to user editing actions should not, however, noticeably degrade the perceived editing response.

During execution, we will exploit the availability of the program source representation and the high-bandwidth user interface (such as a multi-window high-resolution display screen) that is typical of integrated environments to provide the user with useful feedback on the progress of program execution. Such feedback aids user comprehension of program behavior and thereby helps reduce debugging time. The Cornell Program Synthesizer [Teitelbaum81] provided many of the features we believe should be present in an IPE.

Of course, we have committed ourselves to the idea of *generating* such environments, rather than hand-coding them. The Synthesizer Generator [Reps84] and other systems demonstrated that it is possible to generate the syntactic and static semantic parts of an IPE from formal descriptions. Although there have been attempts to do the same for the dynamic semantic aspects of languages, these attempts have not been as successful, for various reasons.

The related field of *semantics-directed compiler generators*, however, has seen much recent work, based primarily on denotational semantics. Systems like SIS [Mosses76] and PSP [Paulson82] as well as the more recent work of Sethi [Sethi83] and Appel [Appel85] demonstrate the feasibility of generating language processors from denotational definitions.

There are three issues that must be addressed in order to adapt compiler-generation techniques to integrated programming environments:

- The unacceptable performance that is typical of compilers generated from denotational semantics.
- The requirement for *incremental translation*, which can be used to simulate *instant* translation, thus contributing to the appearance of integration.
- The need to provide support for *interactive execution*, so as to exploit the special nature of an integrated programming environment to speed up the debugging process.

This thesis explores an approach to the generation of execution facilities for integrated programming environments based on a denotational semantics of the object language. Our approach, called EDS (for Executable Denotational Specifications), translates programs into an internal executable representation that corresponds very closely to the parse tree of the program. In this representation, each node represents the denotation of a particular syntactic construct, and the nodes are threaded in accordance with the flow of control of the program.

Several benefits accrue from the use of this executable representation. Firstly, the syntactic correspondence permits easy incremental translation during editing without degrading response. Secondly, the use of compiled code within denotations provides reasonable execution speed, which is further augmented by the use of realistic representations of stores and environments, in contrast to the more general function representation used in many compiler generators based on denotational semantics. Finally, code sharing between denotations keeps the executable representation compact — apart from a single copy of each distinct denotation function (one for each production in the syntax of the language), the additional storage required for the executable representation consists of a few pointers per node in the parse tree of the program.

The correspondence between the executable representation and the parse tree of the program can be exploited to permit the user to interact with the program at run-time in terms of source-language constructs, thus providing support for interactive execution. Many of the features provided in hand-coded IPEs can be incorporated naturally into the denotational specification of the language, thus providing a convenient way to experiment with these features in IPEs for various languages.

Overall, we were able to achieve execution speeds comparable to that of hand-coded interpreters such as **px**, the Berkeley Pascal interpreter, for simple programs. We argue that this performance can be maintained for all the control flow aspects of languages like Pascal.

Other aspects of Pascal are more troublesome; we do not anticipate that an implementation based purely on denotational semantics will be able to attain performance comparable to hand-coded interpreters for languages with type, scoping and environment systems as complex as Pascal. For this reason, we believe that the ideal integrated environment generation system will be a hybrid, employing a method such as an attribute grammar to handle these aspects of the language, combined with a denotational semantics that embodies the control flow aspects of the language.

The question of how best to obtain such a hybrid language specfication for a language is a question that is not answered in this thesis, but one that we believe needs to be addressed; the final chapter of this thesis outlines some possible approaches to the problem.

Chapter 2

Denotational Semantics

This chapter provides a tutorial introduction to denotational semantics. Readers familiar with the subject may wish to skip this chapter. Those wishing to study the subject further, especially the more theoretical aspects of it, should consult a book on the subject, such as [Stoy77].

There are really two separate aspects to denotational semantics: the techniques used to model programming language constructs, developed primarily by Christopher Strachey; and the theoretical foundations that ensure that these techniques work, developed primarily by Dana Scott. Since this thesis is concerned more with the use of denotational semantics to define (and implement) languages and less with the underlying theory (although we are grateful for its support), our introduction will reflect this concern.

2.1. The structure of a denotational definition

A denotational semantics for a programming language defines a mapping from programs in the language to mathematical objects that represent their meanings. This mapping is defined by structural induction on the syntax of the language; the meanings of simple constructs are defined directly, and the meanings of composite constructs are defined in terms of the meanings of their syntactic constituents. The mathematical object representing the meaning of a construct is referred to as its **denotation**, and hence the term *denotational*.

In addition to the syntax of the language, a denotational semantics specifies the **semantic domains** to which denotations belong, as well as the **semantic functions** that define the mapping from syntactic entities to their denotations.

2.1.1. The semantic domains

Specifying the domains that contain the denotations of language elements is an essential part of a denotational definition. We use domains rather than sets in order to avoid certain mathematical problems that arise from the use of recursive definitions of functions and domains. Although our earlier example of binary numerals (Figure 1-3 in section 1.1.2.3) did not require recursive definitions, they are needed to model realistic programming languages. The reader interested in the theoretical difficulties that arise and their solution is referred to [Stoy77]; we state here without proof that permissible domains are countably based, continuous complete lattices, and that allowed functions on domains are continuous.

Fortunately for those interested in denotational semantics as a tool for language definition rather than as a theory in itself, it is fairly easy to ensure that the semantic domains used are theoretically sound. The way to do so is to allow the definer to specify only a restricted class of domains. In particular, the domains specified should be either **primitive** domains, or constructed from primitive domains using a limited set of domain **constructors**.

Primitive domains are either **standard** domains or explicitly specified **finite** domains. Standard domains include **Int**, the domain of integers; **Bool**, the domain of truth values; and **Id**, the domain of identifiers. Finite domains are defined by listing their elements.

Domain constructors are used to build domains from other domains. The domain constructors usually allowed are:

Product The product domain ($\mathbf{D_1 \times D_2 \times ... \times D_n}$) is the domain of all n-tuples ($\mathbf{d_1, d_2, ..., d_n}$) of elements $\mathbf{d_1 \in D_1}$, $\mathbf{d_2 \in D_2}$, ..., $\mathbf{d_n \in D_n}$. *Selectors* may be used to cull a particular element from a tuple.

Sum The sum domain ($\mathbf{D_1 + D_2 + ... + D_n}$) consists of elements of any one of $\mathbf{D_1, ..., D_n}$, where each value has associated with it a *tag* indicating which domain it came from. The use of a tag distinguishes the sum from a union. The tag is useful in the case where there are elements that are common to more than one component of a sum. *Injection* and *projection* operators, together with tag tests, allow elements to be inserted into and extracted from sum domains.

Function The function space ($\mathbf{D_1 \rightarrow D_2}$) is the domain of *continuous* functions from $\mathbf{D_1}$ to $\mathbf{D_2}$. We will not define what is meant by *continuous* functions, except to say that they preserve the structure of domains. The notation we will use only allows the definition of continuous functions. Since all computable functions are continuous, this restriction does not affect the power of our method.

The domains used in a denotational definition are specified by means of a system of **domain equations**,

$$\mathbf{D_1 = rhs_1}$$
$$\vdots$$
$$\mathbf{D_n = rhs_n}$$

where $\mathbf{D_1, ..., D_n}$ are the domains being defined and each right-hand side $\mathbf{rhs_i}$ is a

domain expression containing primitive domains, perhaps some of the $\mathbf{D_j}$, and the domain constructors defined above.

The desired domains $\mathbf{D_1}$,...., $\mathbf{D_n}$ are then a solution of this system of domain equations. For technical reasons, we must interpret the "=" in the equations as signifying *isomorphism* rather than strict equality.

If a system of equations can be ordered so that for all i, rhs_i contains only primitive domains and $\mathbf{D_1}$ ... $\mathbf{D_{i-1}}$, then the system is said to be **non-recursive**, and a solution can easily be found through a process of repeated back substitution, starting with $\mathbf{D_1}$, which is defined in terms of primitive domains exclusively.

If it is not possible to order a system of equations as described above, then the system is said to be **recursive**. Non-recursive systems are not powerful enough to define the domains needed to specify the semantics of actual programming languages. Unfortunately, recursive equation systems are not as straightforward to solve as non-recursive ones. In particular, recursive equation systems can give rise to an *uncountable infinity* of solutions. In such cases, we take the *least* solution, that is, one that is isomorphic to a subset of any other solution. The restrictions we have placed on the structure of domains guarantee the existence of a unique least solution.

Finally, note that our definition of domain equations would permit equations of the form

$$\mathbf{D = ... + D \rightarrow D + ...}$$

If we were considering sets, a solution to this equation would have the strange property that a subset of itself would be isomorphic to its own function space. If a set has more than one element, the cardinality of its function space is strictly greater than that of the set itself, precluding such an isomorphism. Fortunately, we are considering *domains*

rather than sets, and restricting ourselves to *continuous* functions, so the above argument does not affect us.

2.1.2. The semantic functions

Once we have specified the abstract syntax of the language and the semantic domains into which language constructs will be mapped, it remains only to specify the mapping, the *semantic functions*. There is one semantic function for each syntactic category in the language, defined by cases on the productions in the abstract syntax for that category. The specification of a semantic function includes the syntactic category to whose elements it assigns denotations, as well as the domain to which those denotations belong. In most cases, this domain is a function space.

The language in which semantic functions are written is called the *metalanguage* of the semantic description. This metalanguage is usually the λ-calculus. Use of the λ-calculus enables us to specify all (and only) the computable functions, and permits easy manipulation of higher-order functions.

2.2. A Simple Language

We will now define L, a simple programming language, as a vehicle for introducing some of the techniques of denotational semantic specification. Figure 2-1 gives the syntax of L.

2.2.1. Informal Semantics of L

This sub-section describes the semantics of L informally, in the manner of a language definition or user manual.

```
<prog>::=   Program ( id ) <decl> <stmt>
<decl> ::=
<decl> ::=  <decl>1 <decl>2
<decl> ::=  const id = <expr>
<decl> ::=  var id
<stmt> ::=
<stmt> ::=  <stmt>1 <stmt>2
<stmt> ::=  id := <expr>
<expr>::=   numeral
<expr>::=   id
<expr>::=   <expr>1 + <expr>2
```

Figure 2-1: Syntax of L.

2.2.1.1. Programs

The header of a program specifies a particular identifier whose value at the end of program execution is the meaning of the program. If this identifier is not declared in the program, the program is erroneous. The body of a program consists of a sequence of declarations followed by a sequence of statements. The declarations are in effect during the execution of all the statements in the program; there is no block structure. There is only one data type in L; all variables and expressions are of type integer.

2.2.1.2. Declarations

Declarations can be constant declarations, which bind identifiers to values, or variable declarations, which merely set aside storage without assigning a value. The expression that defines the value of a constant may use constants declared previously; use of variables or undeclared identifiers is erroneous. It is illegal to have more than one declaration for an identifier.

2.2.1.3. Statements

For now, the only statements allowed in L are assignment statements, which evaluate the expression on the right hand side and store its value into the memory location denoted by the identifier on the left hand side; this identifier must have been declared as a variable. Sequences of statements are executed in textual order.

2.2.1.4. Expressions

Numerals

<expr>::= numeral

A numeral is the simplest form of expression. Its value is the corresponding number.

Addition

<expr>::= $<expr>_1$ + $<expr>_2$

An expression can be the sum of two expressions, and its value will be the sum of the values of those expressions. We could as easily include other operators than +, but do not for reasons of brevity.

Identifier reference

<expr>::= id

The use of an identifier in an expression yields the value it denotes. If the identifier has been bound to a value via a constant declaration, then the bound value is used; if it has been declared as a variable, then the value most recently assigned to it is used. If the identifier has not been declared, or is a variable that has not yet been assigned a value, the program is erroneous.

2.2.2. Denotational Semantics of L

We now proceed to give a denotational semantics for L.

2.2.2.1. The semantic domains

2.2.2.1.1. Basic Values

In our original example of binary numerals, the value domain was simplified by the fact that every binary numeral denotes an integer. This is not the case in L, where syntactically well-formed expressions may contain semantic errors, and hence not denote any number. Examples of semantic errors would include division by zero (if L were to include a division operator) and the use of an undefined identifier. Consequently, we define our value domain to be

$$\mathbf{Val} = \mathbf{N} + \{\bot\}$$

where $\bot$ (bottom) is a special error value, distinct from all elements of **N**.

In conjunction with this definition of **Val**, we extend our basic functions on **N** (notably +) to **Val** by making them **strict**. In other words, the function + for elements of **Val** has the value $\bot$ if either of its arguments is $\bot$, and is otherwise identical to the function + for elements of **N**.

2.2.2.1.2. Environments

At first glance, it appears that *E*, our (yet to be defined) meaning function for expressions, should map expressions to elements of **Val**. However, the value of an expression containing an identifier, such as "x + 1", can be determined only when we know the context in which it is to be evaluated (in this case, the value to which x is bound). We have said that our definition of the meaning of "x + 1" must be given denotationally, that is in terms of the meanings of its components only. This require-

ment precludes the use of the bound value of x, since the binding of this value is not part of the expression to which a meaning is being ascribed.

The solution to this problem is to define the meaning of an expression to be a **function** from a collection of variable bindings to a value, rather than a value in itself. We refer to the collection of variable bindings as an **environment**, and introduce the domain of environments:

$$\mathbf{Env} = \mathbf{Id} \rightarrow \mathbf{Val}$$

The meaning function E for expressions can then be defined to have functionality

$$E : \langle\text{expr}\rangle \rightarrow \mathbf{Env} \rightarrow \mathbf{Val}$$

This says that E maps expressions (members of <expr>) into functions that map environments (members of **Env**) into the value of the <expr> in that environment.

2.2.2.1.3. Stores

Although not strictly required for defining the semantics of L, it is convenient to introduce the concept of a **store** at this point. As its name suggests, a store is used to model the memory of a computer. We define the domains **Loc** and **Store**, which model memory addresses and memory itself.

$$\mathbf{Loc} = \mathbf{N} + \{\perp\}$$

$$\mathbf{Store} = \mathbf{Loc} \rightarrow \mathbf{Val}$$

The value $\perp$ as a possible element of **Loc** allows the modeling of invalid addresses, such as variables for which storage has not been allocated. In conjunction with the introduction of the **Store** domain, we modify the domain **Env** to map identifiers to locations as well as values,

$$\mathbf{Env} = \mathbf{Id} \rightarrow (\mathbf{Val} + \mathbf{Loc} + \{\textit{undefined}\})$$

An environment will map an identifier to a value if the identifier is bound to a constant value (via a **const** definition), to a location if the identifier denotes a variable, and to

undefined otherwise. The use of a two-stage mapping (**Id** $\rightarrow$ **Loc** and then **Loc** $\rightarrow$ **Val**) will allow the modeling of programming language constructs such as pointers, reference parameters, and other situations where two or more identifiers can be *aliases* of the same memory location (and hence the same value).

The meaning of an expression now requires both an environment and a store to yield a value, so that the semantic function for expressions becomes:

$$E : \text{<expr>} \rightarrow \textbf{Env} \rightarrow \textbf{Store} \rightarrow \textbf{Val}$$

2.2.2.2. The semantic functions

Having defined our semantic domains, we are now ready to define the semantic functions that ascribe meaning to the constructs of L. We will have one semantic function for each of the syntactic categories in L, namely <prog>, <decl>, <stmt> and <expr>.

2.2.2.2.1. *E*: The semantic function for expressions

We will first define *E*, the meaning function for members of the syntactic category <expr>. As stated above, the functionality of *E* is

$$E : \text{<expr>} \rightarrow \textbf{Env} \rightarrow \textbf{Store} \rightarrow \textbf{Val}$$

E is defined by cases on the possible productions for <expr>, as in Figure 2-2. Note that we are not using λ-calculus to define the *E*, but a notation similar to that used in [Tennent81]. The variables env and store represent elements of the domains **Env** and **Store**, respectively. Enclosing syntactic constructs in brackets signifies the application of a meaning function to them. In cases where there is only one function that can be applied, the function name will be omitted. Thus, [numeral] on the right hand side in Figure 2-2 represents the meaning function for numerals applied to numeral. This meaning function is left undefined here, but is understood to map numerals into

the numbers they denote; similarly, the meaning function for identifiers is assumed to map them into elements of the semantic domain **Id**. The notation foo ? bar is used to test whether foo (an element of a union domain) is a member of domain bar. The /* and */ delimit comments.

Note that the definition of *E* [id] tests the domain of env([id]) to determine whether it is a constant (with a directly available value) or a variable (which must be mapped to a value through the store). It is assumed that store is strict, in that if the location addressed is ⊥, the value returned will also be ⊥. By default, all semantic functions will be strict, so if a semantic function definition does not explicitly state what the result will be when one of its arguments is ⊥, it should be assumed to be ⊥. Note also that we have used the function + defined on the domain (**Val** × **Val**) → **Val** rather than on the integers, as alluded to when we introduced the domain **Val**.

Finally, observe that the semantic definition of a sum expression specifies that both component sub-expressions are to be evaluated in the context of the initial environment and store, thus making the evaluation order of the sub-expressions

---

E [numeral] env store = [numeral]

E [id] env store =
if env([id]) = *undefined* then ⊥
elsif env([id]) ? **Val** then env([id])
else /* env([id]) belongs to **Loc** */
store(env([id]) /* could be ⊥ */

E [<expr>$_1$ + <expr>$_2$] env store
= *E*[<expr>$_1$] env store + *E*[<expr>$_2$] env store

Figure 2-2: Definition of *E*.

---

irrelevant.

2.2.2.2.2. *C*: The semantic function for statements

We call the meaning function for statements *C* (Figure 2-3), following the literature, where statements are often referred to as "commands." Commands affect the contents of memory, so they are *store transformers*. In addition, command execution can result in an error (such as assigning to a constant), which would yield ⊥ as the meaning of the command. We define the domain of command results to be:

$$\textbf{Res} = \textbf{Store} + \{\bot\}$$

The functionality of *C* is then:

$$C : \langle\text{stmt}\rangle \rightarrow \textbf{Env} \rightarrow \textbf{Store} \rightarrow \textbf{Res}$$

That is, the denotation of a statement is a function that takes an environment and store and produces an updated store, or results in an error.

```
C [ ] env store =       store   /* Null command has no effect */

C [ <stmt>1 ; <stmt>2 ] env store
              =         if C [ <stmt>1 ] env store = ⊥ then ⊥
                        else C [ <stmt>2 ] env store'
                                where store' = C [ <stmt>1 ] env store

C [ id := <expr> ] env store
              =         if E [ <expr> ] env store = ⊥ then ⊥
                        elsif env([id]) ? Loc then
                                update(store, env([id]), val)
                                        where val = E [ <expr> ] env store
                        else ⊥
```

Figure 2-3: Definition of *C*.

The primitive function **update** takes a store, a location, and a value, and produces an updated store. It has functionality

update : Store → Loc → Val → Store

and is, as always, strict.

2.2.2.2.3. *D* : The semantic function for declarations

D, the meaning function for declarations, is given in Figure 2-4.

Since declarations can update both environments and stores (by allocating space for variables), and can cause errors (through attempted re-declaration), the result of

---

```
D [ ] env store =        ( env × store )

D [ <decl>1 ; <decl>2 ] env store
              =          if D [ <decl>1 ] env store = ⊥ then ⊥
                         else D [ <decl>2 ] env' store'
                                 where ( env' × store' ) = D [ <decl>1 ] env store

D [ const id = <expr> ] env store
              =          if env([id]) = undefined then
                                 if E[ <expr> ] env store = ⊥ then ⊥
                                 else (env' × store)
                                          where env' = env[[id] ← val]
                                                   where val = E [ <expr> ] env store
                         else ⊥  /* Re-declaration */

D [ var id ] env store
              =          if env([id]) ≠ undefined then ⊥
                         elsif allocate(store) = ⊥ then ⊥
                         else (env' × store')
                                 where env' = env[[id] ← loc]
                                 and (store' × loc) = allocate(store)
```

Figure 2-4: Definition of *D*.

---

declarations is in the domain

$$\textbf{DRes} = (\textbf{Env} \times \textbf{Store}) + \{\perp\}$$

The functionality of *D* is then:

$$D : \langle\text{decl}\rangle \rightarrow \textbf{Env} \rightarrow \textbf{Store} \rightarrow \textbf{DRes}$$

The notation env[foo ← bar] denotes an environment that is identical to env for all argument values except foo, where it has value bar. We also introduce the primitive function **allocate**, which allocates a free location from store. If there is no free location, **allocate** returns ⊥, otherwise it returns the allocated location and a new store. Note that we have not constrained **allocate** to use any particular storage allocation discipline. **Allocate** has functionality

$$\textbf{allocate} : \textbf{Store} \rightarrow (\textbf{Store} \times \textbf{Loc}) + \{\perp\}.$$

2.2.2.2.4. *P* : The semantic function for programs Finally, Figure 2-5 defines *P*, the semantic function for programs, which has functionality

$$P : \langle\text{prog}\rangle \rightarrow \textbf{Val}.$$

In the definition, env_0 denotes the predefined environment (for L, this maps all identifiers to *undefined*) and store_0 denotes the initial memory, which is completely unallocated.

2.3. Extending L

Given the domains and techniques introduced in the previous section, we can extend L to include more expression and statement types. The interested reader is referred to [Tennent81] or [Gordon79] for more examples.

There are also common programming language constructs that cannot be modeled using the techniques described thus far. Some of these constructs are examined below.

P [**Program** (id) <decl> <stmt>]
= if D [<decl>] env_0 $store_0$ = ⊥ then ⊥
elsif env([id]) = *undefined* then ⊥
where (env × store) = D [<decl>] env_0 $store_0$
elsif C [<stmt>] env store = ⊥ then ⊥
where (env × store) = D [<decl>] env_0 $store_0$
elsif env([id]) ? **Val** then env([id])
elsif env([id]) ? **Loc** then store'(env([id]))
where store' = C [<stmt>] env store
else ⊥

Figure 2-5: Definition of *P*.

2.3.1. Iteration

One interesting extension to L that we will discuss here is the **while** statement,

<stmt> ::= **while** <expr> <stmt>

which has the usual meaning: <expr> is evaluated, and if it has non-zero value, <stmt> is executed and then <expr> is evaluated again. The loop is exited when <expr> evaluates to zero.

2.3.1.1. An Incorrect Definition A first attempt to give a denotational definition for the while statement might result in something like Figure 2-6. The problem with this definition is the use of the *entire* while statement on the right hand side of the definition, which destroys the structural induction; instead of defining the statement in terms of its components, we are attempting to define it in terms of itself. The problem with this becomes obvious when we consider a non-terminating loop where the controlled statement does not affect the store. The "definition" in Figure 2-6 then reduces to

C [**while** <expr> <stmt>] env store = C [**while** <expr> <stmt>] env store

since the execution of <stmt> does not change the store. This definition is circular, and tells us nothing about the meaning of the loop, which we would like to define to be erroneous.

2.3.1.2. The Solution

Although our "definition" above does not necessarily define anything, it does hold as an *equation*. We can use this equation to derive a fixed-point definition of the meaning of the while loop, but the following derivation, adapted from [Demers82] is perhaps more intuitive, if less formal. Note that the program fragment

if <expr> **then**
    <stmt>
    while <expr> <stmt>

where one iteration of the loop has been "unrolled," is equivalent to the original loop. Define the abbreviations

WHILE ≡ **while** <expr> <stmt>

and, for any stm,

IF(stm) ≡ **if** <expr> **then** <stmt> ; stm

and the shorthand notation

$IF^i(stm) \equiv IF(IF(...IF(stm)...)) \qquad i \geq 0$

C [**while** <expr> <stmt>] env store
= if *E* [<expr>] env store = ⊥ then ⊥
elsif *E* [<expr>] env store = 0 then store
else *C* [**while** <expr> <stmt>] env store'
where store' = *C* [<stmt>] env store

Figure 2-6: Attempted definition of while statement.

---

where IF has been applied i times. Then the equivalence noted above can also be written as

$$\text{WHILE} = \text{IF(WHILE)}$$

which indicates that WHILE is a "fixed point" of the statement transformation function IF. We can also see that WHILE is the *least* fixed point, by considering the statement

$$\text{IF}^i \equiv \text{IF}^i(\textbf{abort}) \qquad i \geq 0$$

where **abort** is a special statement that always produces the result $\perp$. If IF^i produces a result other than $\perp$ then it produces the same result as WHILE. In addition, whenever WHILE terminates, it does so after finitely many steps, and thus produces the same result as IF^j for some j. In fact, if WHILE terminates after k steps, then it produces the same result as IF^j for all $j > k$, so that WHILE is the *limit* of IF^j as $j \rightarrow \infty$. If WHILE does not terminate, then IF^j produces $\perp$ for all j, and its limit is trivially $\perp$, which again is the result of WHILE.

We can thus define the semantics of a while loop as a limit (Figure 2-7). This

---

C [**while** <expr> <stmt>] env store
= $\lim_{i \rightarrow \infty} p_i$(env store)
where $p_0(e\ s) = \perp$
and $p_{i+1}(e\ s)$
= if E[<expr>] e s = $\perp$ then $\perp$
elsif E[<expr>] e s = 0 then s
elsif C[<stmt>] e s = $\perp$ then $\perp$
else $p_i(e\ s')$
where s' = C[<stmt>] e s

Figure 2-7: Limit definition of while loop.

---

limit may not always be computable, but it is computable for all finite loops, and some infinite ones.

2.3.2. Procedure calls

Non-recursive, parameterless procedures with open scoping and no local definitions are the easiest to model. In this case, a procedure is simply a list of commands, and therefore represents a state transformation in the domain

$$\textbf{Proc} = \textbf{Store} \rightarrow \textbf{Store} + \{\perp\}$$

Procedure names can be mapped to bodies by extending the environment mapping to include **Proc** as a component of the range.

Defining the semantics of procedure calls that involve recursion is more difficult. We will not give the details here, but the approach is similar to that described above for the **while** loop; call sequences of increasing (but bounded) depth can be used to approximate the effect of recursive calls of (potentially) unbounded depth, and the meaning of unrestricted calls is then the limit of these (progressively better) approximations.

Parameters, local variables, and scoping rules all make the definition more complex, but do not require the introduction of any concepts beyond those already described. The interested reader is referred to the literature on the subject [Stoy77, Gordon79, Tennent81].

2.4. Continuations

Unstructured constructs, especially those that cause non-local transfers of control, such as **gotos**, are hard to model in the denotational framework described thus far. The problem is that the meaning of a construct is assigned *compositionally*, in terms of the meanings of its components, and the meaning of these components is determined

independently of each other. Consider a **stop** statement, whose semantics are to discontinue execution and return the current store as the final result of the program. Unfortunately, this affects the semantics of the rest of the statements in the program, which now (as part of a program containing a preceding **stop** statement) have no effect at all on the computation. Since we require that the meaning of a statement be assigned independent of context, we cannot have different semantic functions for statements depending on whether or not a **stop** statement has been encountered.

One way to model a **stop** statement is to include a special flag in the state of the computation that tells us whether we have encountered a **stop** during execution. The denotations of statements will then check this flag and return the current store unaltered if it has been set. Although this solution works, it is not particularly elegant, and does not extend cleanly to other non-local transfers of control, such as statements of the form **goto** id, where id is an arbitrary program label.

The notion of a **continuation** was developed by Christopher Wadsworth [Strachey74] to describe the semantics of control transfers. In a semantics using continuations, an extra parameter, the continuation, is supplied to the denotation of every command. The continuation is a function that represents the meaning of the remainder of the program at that point. The domain of continuations is usually of the form

$$\mathbf{Cont} = \mathbf{Store} \rightarrow \mathbf{Ans}$$

where **Ans** is the domain of answers, or program results (usually part of the store, or the contents of an output file). Thus the functionality of *C*, the semantic function for commands, now becomes

$$C = \langle\text{stmt}\rangle \rightarrow \mathbf{Env} \rightarrow \mathbf{Cont} \rightarrow \mathbf{Store} \rightarrow \mathbf{Ans}$$

Usually, a command produces an updated store and then passes that as an argument to

its continuation, which returns the final answer of the program. However, the command is free to ignore the supplied continuation, and control transfers will do just that. The denotation of a **stop** statement, for example, will return the appropriate component of the current state directly as the result of execution. A statement of the form **goto** id can look up "id" in the environment (which will now contain identifier→continuation bindings) and invoke the bound continuation rather than the supplied parameter.

The introduction of continuations also simplifies error handling. Erroneous constructs can return *error* immediately as the result of the computation, and the semantic functions need not be concerned with propagating input error values.

Although we have introduced continuations in the context of *commands*, we can have other types of continuations. Declaration continuations, for example, would map the results of declarations (belonging to the domain **Env** × **Store**) into the domain of answers. Figure 2-8 gives the semantics of some sample constructs from L using continuations. Note the use of the keyword "rec" in the definition of the semantics for a labeled statement. This indicates that the definition of cont' involves cont' itself, and hence should be determined as the fixed point of an appropriate functional. This example limits the scope of a label to the labeled command. More general label scoping is possible, with attendant complexity in the definition of the semantics of labeled commands, but is beyond the scope of this introduction. The interested reader may consult [Stoy77] for details.

2.5. Summing up

The concepts introduced thus far are sufficient to model the semantics of languages such as Pascal [Jensen78], whose denotational semantics are given in [Tennent78]. It is not possible within the constraints of this thesis to expatiate on

Semantic Domains

Env = **Id** → (**Val** + **Loc** + **CCont**)
CCont = **Store** → **Ans**
ECont = **Val** → **Ans**

Semantic Functions

C [ ] env cont store = cont(store)
C [<stmt>$_1$ <stmt>$_2$] env cont store
= *C* [<stmt>$_1$] env cont' store
where cont'(s) = *C* [<stmt>$_2$] env cont s
C [id := <expr>] env cont store
= *E* [<expr>] env k store
where k(n) =
if adr ? **Loc** then cont(**update**(store, adr, n))
else ⊥
where adr = env([id])
C [**goto** id] env cont store
= if target ? **CCont** then target(store) else ⊥
where target = env([id])
C [**label** id : <stmt>] env cont store
= cont'
where rec cont' = *C* [<stmt>] env[[id] ← cont'] cont store

Figure 2-8: Semantics of L using continuations.

the entire subject of denotational semantics, nor would it be desirable; many better equipped to do so have already contributed to the literature, and readers seeking to learn denotational semantics would be best advised to begin their quest there. Our goal here is not to contribute to denotational semantics, but rather to use the ideas contained therein to simplify the task of programming environment generation. The aim of this chapter, therefore, was to acquaint the reader unfamiliar with denotational semantics with the tools and techniques of the trade, in order to facilitate their use in later chapters.

In many cases, we will not be concerned with the specifics of a denotational definition of a particular language, but rather with the types of constructs used to model the semantics of a variety of languages. Environments, stores, continuations, and limits are some of the concepts that we will invoke in later discussion, together with ideas relating to domain definition and structure. If the reader previously unfamiliar with denotational semantics recognizes these terms and has some intuitive understanding of their significance, this chapter will have served its purpose.

Chapter 3

Semantics-Directed Translation

A denotational description of a language specifies a translation from programs in the object language to their denotations in the metalanguage. This translation, combined with a method for interpreting terms in the metalanguage, defines an implementation of the object language. This chapter describes our approach to generating language implementations based on denotational semantics.

3.1. Previous Approaches

In order to set the context for discussion of our approach, we briefly outline some previous approaches.

3.1.1. λ-calculus based translation

We can consider the semantic functions of a denotational semantics as rules specifying the translation of language constructs into λ-calculus expressions involving the denotations of the component constructs as sub-expressions. Translation thus proceeds bottom-up, composing λ-expressions until the denotation of the program is obtained as a single large λ-expression.

This λ-expression can be interpreted by reduction to *normal form.* A leftmost reduction strategy is guaranteed to find the normal form if it exists, but is extremely time-consuming. Other reduction strategies, such as innermost first ("call by value") are less powerful but better suited to mechanical reducers, since no β-reduction is performed unless the argument is in normal form.

Peter Mosses's SIS (Semantic Implementation System) [Mosses76] uses the technique just described — it produces the denotation of a program (known as the "semantic parse") and interprets the resulting λ-expression. Larry Paulson's PSP system [Paulson82] includes a *simplifier* that performs certain "compile-time" reductions on the denotation of a program before interpreting it; interpretation is via a stack machine with an innermost reduction order.

The major drawback of λ-calculus based systems is the inefficiency of manipulating functional values such as environments and stores. Naive λ-calculus systems cannot take advantage of the very limited ways in which stores and environments are manipulated in denotational descriptions of common programming languages to perform these manipulations more efficiently. As a result, Pascal programs processed by PSP execute about 1000 times as slowly as programs compiled with a conventional compiler.

3.1.2. Smart semantics processors

Programming language designers often have in mind particular representational tricks or implementations when incorporating features into a language. For example, stack-based name resolution and storage allocation are commonly used in Pascal-like languages. Unfortunately, such assumptions are not always explicit in the semantic specification of the language, so that an automatically generated translator cannot make use of efficient implementations of the underlying concepts, and hence performs poorly compared to a hand-written system.

As a simple example, consider the store in a denotational definition. Programming languages for von Neumann machines invariably assume a single store, which is updated in place (imperatively). Denotational descriptions of these languages, on the

other hand, may contain several occurrences of expressions yielding values in the domain of stores, and several store-typed arguments to semantic functions. In order to implement the language efficiently, it is vital that we be able to replace all these instances of stores by references to a single global store, with appropriate imperative updating operations. Recently, David Schmidt [Schmidt85] developed sufficient criteria for a denotational definition to be correctly transformable in this way. A denotation definition possessing this property is said to be *single-threaded* in the store.

There are, however, many other implementation techniques that are well-known to language designers and compiler writers, yet their applicability to a particular language cannot be deduced from its denotational semantics. One approach to solving this problem is to use a "smart" semantics processor, that knows about concepts such as stores and environments, and provides higher-level operations that can be applied to them. The semantics of languages can then be written using the operations provided, eschewing more general manipulations of the underlying objects, and freeing the semantics processor to use "clever" (and efficient) representations for them.

There is, of course, a drawback to this approach; building in assumptions about the implementation of certain constructs limits the generality of the languages that the semantics processor can handle. Furthermore, the semantics of the object language must be tailored to fit the assumptions made in the semantics processor.

3.1.3. Combinator-based approaches

Several researchers [Wand82, Sethi83] have proposed sets of special purpose *combinators*. These combinators are chosen to resemble closely conventional machine code, usually for a hypothetical stack machine. The denotational semantics is written in terms of these combinators, rather than using general λ-expressions. The denotation

of a program is thus a tree whose nodes are combinators, and can be interpreted to simulate program execution. Suitable choice of combinators allows some transformation of the tree before execution. Since the combinators correspond to conventional machine operations and there are no general λ-expression manipulations, execution is reasonably efficient.

The drawback of the combinator-based approach is again the loss of generality. Sethi's system, for example, can handle the control structures of the C programming language, but not data structures such as records, which require extensive manipulation of environments.

3.2. EDS: Executable Denotational Specifications

We turn now to a description of our system for implementing languages based on a denotational semantics. This system is called EDS (for Executable Denotational Specifications). The implementor rewrites the standard semantics of the object language, transforming it into an implementation-oriented definition that is supplied to EDS. EDS then uses this definition to generate an implementation of the object language.

Although EDS is presented here as a stand-alone interpreter generator, it is intended to be combined with a language-based editor generator to produce a generator for complete integrated programming environments. Chapter 4 describes how interpreters generated by EDS can be adapted to provide the special characteristics desired in run-time facilities of IPEs.

EDS employs the smart semantic processor approach described previously together with a variant of the combinator-based approach to translator generation. Instead of providing a fixed set of combinators, EDS permits the language implemen-

tor to define combinators, which may then be used in the language specification. In addition, "clever" implementations are generated for certain common constructs encountered in denotational specifications.

Semantic descriptions processed by EDS are first-order; they contain no functional domains. Instead of functional domains, EDS uses first-order concrete representations of those domains. Subsequent sections will describe how the functional domains usually encountered can be replaced by first-order representations.

The remainder of this section describes a version of EDS that supports only *direct* semantic specifications, without continuations. Subsequent sections will describe how EDS is extended to handle continuation-style semantics.

3.2.1. The form of an EDS specification

The semantic aspects of the language specification, with which we are concerned here, are divided into three sections: the semantic domains, the combinator definitions, and the meaning functions. Each of these will be described in turn below.

Although denotational semantics are traditionally specified in a *functional* language, this is not the case in EDS; implementations specified in EDS are translated into a general-purpose programming language (Modula-2) [Wirth83], and the language specifier is permitted to write segments of the definition in Modula-2 for efficiency if desired. The EDS specification language is thus a *superset* of Modula-2, with extensions to support the definition and manipulation of domains and combinators, but has a syntax based on that of Modula-2.

3.2.1.1. The semantic domains

Domain declarations in EDS serve two purposes: to help detect errors in the denotational specification, and to enable EDS to generate appropriate implementations for elements of the domain. The following sub-sections describe how domains are declared in EDS. For the most part, these domain declarations follow the domain construction operations described in chapter 2, modified somewhat for convenient implementation. One notable exception is the function domain constructor, which is not supported by EDS. Instead of functional domains, EDS specifications are expected to use first-order concrete representations of elements of those domains, which take into account the manner in which function-domain elements are manipulated in order to provide efficient implementations for them. Specifically, the provision of an *opaque* domain facility, together with combinator definitions, allows the internal structure of certain domains to be hidden from other sections of the semantic description, thus preserving some abstraction while permitting efficient implementation.

3.2.1.1.1. Predefined domains

Common domains are predefined by EDS. Some, such as **Int** and **Bool**, correspond to the predefined types usually available in programming languages, while others, such as **Id**, occur frequently in denotational definitions. One notable predefined domain is the domain **error**, which contains a single value, *errorval*.

3.2.1.1.2. Domain declarations

Domain declarations are introduced by the keyword DOMAIN, and serve to associate a name with a domain structure. Domain declarations may be recursive, and may be written in any order. There is no precedence specified for the domain constructors; only one type of constructor may be applied in a particular definition. The reason for

this restriction is to prevent the creation of anonymous domains, that is, domains with a structure but no name. Anonymous domains complicate the type-checking rules without adding to the power of the system (although they do make domain specification more concise) and hence are forbidden by the current implementation. The following types of domain declarations are permitted:

Enumeration domains

Finite domains can be specified by listing the elements they contain. Currently there is no overloading of names allowed: elements of enumeration domains must be unique. The syntax of an enumeration domain definition is:

<name> = { <element>, <element> ,..., <element> };

Product domains

Product domains can be formed from any two domains. The first component of the product is referred to as the *head* and the second as the *tail*. The syntax of an enumeration domain definition is:

<name> = <headdomain> * <taildomain>;

Sum domains

Any number of domains can be combined into a disjoint union or sum domain. A tag is maintained with each sum domain element to allow testing and extraction of appropriate elements. The syntax of a sum domain definition is:

<name> = <domain> + <domain> + ... + <domain>;

3.2.1.1.3. Opaque domains

EDS permits the declaration of *opaque* domains, whose structure is not visible to the rest of the semantic specification. These domains are treated as primitive domains

by the type-checker. Opaque domains are similar to opaque types in Modula-2 [Wirth83]; in fact they are implemented as such. An opaque domain is specified by declaring its name as a domain without providing a corresponding domain structure in the domain declaration section. The actual internal structure of the domain is described within an implementation module that is not visible to other sections of the semantics.

Elements of opaque domains may be manipulated using special auxiliary functions that implement the abstract operations desired. These operations are defined in an implementation module for the opaque domain, and have access to its internal representation. Opaque domains permit efficient concrete implementations of functional domains while maintaining abstraction of the other sections of the specification.

3.2.1.2. The combinators

The success of the combinator approach to semantics-directed translation depends greatly on the choice of combinators. In EDS, this choice is largely the responsibility of the language specifier. In order to permit efficient implementation of a range of languages, EDS's design philosophy is to permit as much flexibility as possible in combinator specification. This flexibility is not without price, however: the lack of a fixed combinator set forces specification of combinators from scratch for every language. On the other hand, judicious choice of combinators will allow a wide variety of languages to be specified without combinator rewriting; in a sense, the particular choice of combinators defines an *abstract machine* on which the object language will be implemented.

Combinators are similar to language primitives, except that they are defined by the user. The argument and return types for combinators can be defined either directly,

or through the use of named *combinator classes*, which are a convenient shorthand for defining several combinators that have the same argument and result types. A combinator class definition is introduced by the keyword COMBCLASS and specifies the name and types (domains) of the formal parameters (λ-variables), and the result type. For example, the set of combinators for arithmetic operations on integers might be declared as:

ArithOps = COMBCLASS(op1, op2 : integer) → integer;

Combinator definitions themselves have a form similar to procedure declarations, except that the keyword PROCEDURE is replaced by COMBINATOR, and the procedure header (specifying argument and result information) is replaced by the name of the combinator class to which the combinator being defined belongs.

For example, given the definition of ArithOps above, the combinator "plus" could be defined as:

```
COMBINATOR plus : ArithOps;
begin
        return( op1 + op2 );
end plus;
```

Local variables may be declared within the body of combinators; these variables may have any type or belong to any domain that would be visible in the combinator definition following the standard scoping rules of Modula-2.

The body of a combinator can contain any legal Modula-2 statements. In addition, extensions to support manipulation of domain objects are provided — in particular postfix **.Head** and **.Tail** operators to separate the elements of a product domain, the **?** operator to test which domain a value from a union domain belongs to, and the **DCASE** statement, which performs a switch on the tag of an element from a union domain. The projection operator :: extracts a particular element from a union; thus v::integer would yield an integer result. Static type-checking ensures that v does

indeed belong to a union domain having integer as one of its components; run-time tag checking is also possible (to check that v does indeed contain an integer when the expression is evaluated). Assignment statements of the form

```
foo ::= domname( expression )          -- Note the ::=
```

signify a domain injection, where foo is assigned the value of expression, together with a tag signifying that this value belongs to "domname." Once again, static type-checking ensures that foo is indeed a union capable of containing such values.

3.2.1.3. The meaning functions

The meaning functions specify a translation from the syntactic domain into the meaning domain. The domain to which the denotation of a particular syntactic construct belongs is specified by a DENOTECLASS definition, which is identical in form to a COMBCLASS definition. For example, the class of expression denotations could be defined as:

```
ExprDenotation = DENOTECLASS( U : Env; S : Store ) → Val;
```

Although identical in form to combinator class definitions, DENOTECLASS definitions will be used to define denotations rather than combinators, and denotations are different from combinators, as explained below.

The keyword DENOTATION introduces the specification of a denotation, which also includes the name of the denotation (for use in associating denotations with productions) and its domain (defined in a DENOTECLASS definition). The form of a denotation definition is similar to that of a combinator definition. The body of a denotation differs from that of a combinator in that it may contain references to the denotations of the sub-components of the syntactic entity to which it corresponds; these denotations will be bound as parameters during translation of the object language program into its denotation. As an example, consider the definition of the denotation of a sum

expression:

```
DENOTATION AddExp : ExprDenotation;
begin
        return(plus(EvalE(SynArg$1(U, S)), EvalE(SynArg$2(U, S))));
end;
```

The denotations of sub-components are distinguished from other parameters to denotations because they will be passed "by name" rather than "by value," as the other parameters are. Sub-component denotations are also referenced specially, as can be seen in the definition of AddExp above; the notation SynArg$*n* refers to the denotation of the n^{th} syntactic sub-component of this construct. EvalE is a special macro (defined by EDS) which ensures that the sub-component denotation is evaluated in accordance with its denotation class. The reason for these *Eval* macros (there is one defined for each denotation class) will be explained later, during the discussion on translation of denotations.

The final element of the meaning functions is the association of a particular denotation with a production in the syntax of the object language, along with the binding of denotations of sub-components as parameters. For simplicity, EDS binds denotations for *all* sub-components, and does so in the order in which they occur in the production; thus the specification need only provide the denotation name associated with a particular production — the rest is automatic. The denotation AddExp, for example, would be associated with the production for an addition expression in the syntactic specification of the language, as below:

<expr> ::= <expr> + <expr><<MAKEDENOTATION(AddExp, 2);>>

The numeral 2 indicates the number of syntactic sub-components whose denotations are to be bound; this information could be deduced from the production, but has to be supplied in the original implementation of EDS, which used an existing parser genera-

tor.

3.2.2. Generating an implementation

An EDS specification can be used to generate an implementation of the object language. This implementation takes the form of a Modula-2 program, which consists of several modules. Some of these modules are library modules that implement functions built into EDS or provide run-time support for certain operations; other modules are generated from the specification by EDS. An EDS-generated implementation executes object-language programs by converting them into denotations and then interpreting the denotations. We will refer to the process of generating a language implementation from an EDS specification as *interpreter generation.*

3.2.2.1. Domain Implementations

During interpreter generation, domain specifications are converted into equivalent Modula-2 type declarations. A Modula-2 definition module containing these declarations is generated and used when compiling the Modula-2 representation of the combinators and denotations. Each domain is assigned a domain number, which is used to tag values in sum domains, in tests for domain membership, and to control alternative selection in DCASE statements.

Each type of domain specification in EDS has an analogue in the type system of Modula-2. Primitive domains correspond directly to predefined types in Modula-2, with the exception of **error** and **Id**. **Error** is implemented as an enumeration type with a single value, and **Id** is implemented as a string type, with string creation, assignment, and comparison operations. Enumeration domains have an obvious implementation as Modula-2 enumeration types. Product domains are represented as records with two fields, corresponding to the head and tail components of the product. Sum

domains are modeled by discriminated union types, that is variant records with tag fields. The tag field takes on domain number values corresponding to the domains comprising the union, and there is one variant for each component of the union. Opaque domains correspond directly to opaque types in Modula-2; in the definition module, only the type name is specified. The corresponding actual type declaration is the responsibility of the language specifier (although a missing declaration will be noted and cause an error message).

One significant difference between types in Modula-2 and domains in EDS is recursion; EDS domains may be defined recursively, whereas Modula-2's type system does not permit recursion without the definition of an intermediate pointer type. A declaration for this pointer type is supplied where necessary during interpreter generation.[1]

3.2.2.2. The executable representation

During interpreter generation, the denotations specified for object-language constructs are translated into Modula-2 routines, which are then compiled into machine code (also at interpreter generation time). The generated interpreter builds the denotation of an object-language program bottom-up by binding the denotations of sub-components as arguments to the denotation function for the particular production being parsed. After the entire source program has been parsed, its denotation has been created as a deeply nested function call, where some arguments of each call (the denotations of sub-components) have been supplied, while other arguments (the stores and environments, for example) have not been bound.

---

[1] The initial implementation simplified matters somewhat by declaring pointer types for *all* domains that might be recursive, specifically all union and product domains. It is reasonably straightforward to optimize away unnecessary indirection.

The denotation function for a particular construct (say an addition expression) may occur many times in the denotation of a program, with different arguments (sub-expression denotations) each time. To conserve space, an EDS-generated interpreter will maintain only one copy of a particular denotation function, and use pointers to it in the denotation of the program. The denotation of a program is thus a tree that matches exactly the syntax tree of the program, with each node in the tree labeled by (a pointer to the machine-code representation of) a denotation function. The children of a particular node in the denotation are labeled with the denotation functions corresponding to the syntactic constructs that are the children of the corresponding node in the parse tree. Figure 3-1 shows the tree that would represent the denotation of the sequence of assignment statements:

foo := 1; bar := bar + 1;

The function names *CmdSeq, Assign, ConstExp, AddExp,* and *VarExp* represent pointers to the bodies of the appropriate denotation functions. These functions are not defined here, but their intuitive semantics should be obvious; section 3.2.3 defines a complete language, including possible implementations for these denotation functions. Items within quotes, such as "foo" and "1" represent terminal symbols whose denotations are defined directly.

Given this representation of program denotations, together with the association of denotation functions with productions in the syntax of the object language, it is easy to see that translation of a program into its denotation is a simple matter, analogous to constructing the parse tree of a program. How to generate denotations that properly implement the language is not so straightforward; the following section will explain the issues involved and how they have been addressed.

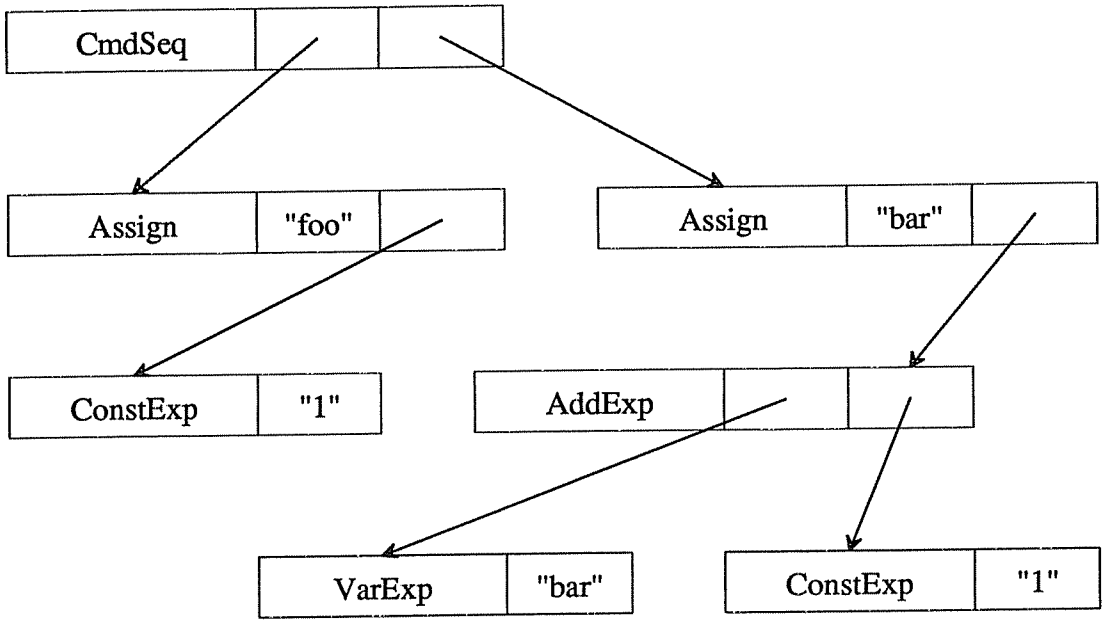


Figure 3-1: Denotation for sequence of assignment statements.

3.2.2.3. Manipulating functions

Denotational definitions manipulate functions extensively. In particular, the ability to define *higher-order* functions, which take functions as arguments and produce functions as results, is required. A denotational semantics also uses "curried" functions, which can be applied to arguments one at a time to produce partially applied intermediate functions, which can then be passed as parameters, assigned, and so on, before being applied to more arguments.

The language used to implement denotations in EDS is Modula-2, which does not treat functions as first-class objects. Modula-2 supports procedure types and variables, as well as formal procedure parameters, but provides no facility for currying or partial

evaluation of functions. We have side-stepped the problem of perturbing functions such as environments by forbidding the definition of explicit function-space domains. Instead, we use first-order concrete representations of elements of function domains together with combinators that update values and control function evaluation. However, we must still deal with the partial application of functions such as denotations to incomplete argument lists. In many cases, the arguments themselves will also be denotations that have been partially applied.

Partially applied functions are represented by a pointer to the function body together with a list of arguments that have already been supplied (some of which may be partially applied functions). In the initial implementation of EDS, only one intermediate stage of partial application is supported; subsequent application must complete the argument list and permit evaluation. When the full argument list is available, the function body pointer (actually a Modula-2 procedure variable) is invoked with the appropriate arguments.

It is possible, indeed likely, that a function will have as a parameter another partially applied function. Such arguments are represented as described above, by a function pointer together with a partial argument list. This representation is similar to the *thunk* mechanism used to implement *by-name* parameters. During the execution of the function, references to the argument function that supply a completing argument list will be expanded into actual calls, just as references to by-name parameters invoke execution of the thunk.

3.2.2.4. Type checking

The use of Modula-2, a strongly typed language, together with the mapping chosen between domain definitions and type declarations, permits extensive type

checking of language specifications at interpreter generation time. This checking speeds the detection of many specification errors, which often manifest themselves as domain typing errors. Although the underlying Modula-2 compiler will detect type errors in a specification, the error messages produced may be hard to relate to the original EDS specification. For this reason, the EDS processor performs some type checking of its own during interpreter generation.

Firstly, all domain definitions are analyzed for consistency. Undefined or multiply defined domains are detected and appropriate error messages are printed. During processing of combinator and denotation definitions, all identifiers occurring in contexts where a domain name is expected are checked to ensure that they do indeed denote a domain. In addition, wherever a particular domain type is required, such as in a domain injection (where a sum domain is required), the specified domain is checked to ensure that it has the required structure.

The Modula-2 type system is used to detect inappropriate injection and projection operations, as well as incorrect usage of .Head and .Tail operators.

3.2.2.5. Translation of combinators

COMBCLASS definitions cause two actions to occur at interpreter generation time: firstly, the entire definition is associated with the combinator class name and stored in a symbol table for later use in processing combinator definitions, and secondly, a Modula-2 procedure type declaration is generated. This declaration follows the COMBCLASS declaration very closely, except that procedure types in Modula-2 do not specify the names of the formal parameters.

COMBINATOR definitions are translated to Modula-2 procedure declarations. The combinator class identifier specified is first checked to ensure that it is indeed

associated with a combinator class definition, and this definition is then used to generate the procedure header. The body of the combinator is translated directly, except for extended operations, such as domain tag testing, injection and projection, and DCASE statements. The Modula-2 analogues of these operations are straightforward given the representation of domain elements chosen.

3.2.2.6. Translation of denotations

DENOTECLASS definitions are processed similarly to COMBCLASS definitions; the difference is that the equivalent Modula-2 procedure type declaration will include additional formal parameters corresponding to sub-component denotations. Since denotations belonging to the same denotation class will, in general, have different numbers of syntactic components, these additional parameters are specified as an open array of unknown size. In addition, the *type* of the sub-component denotations will differ among denotations belonging to the same class, and so the corresponding parameters are declared to be of a type unifying all denotation classes. However, a particular denotation "knows" the denotation class to which the denotations of its sub-components belong, since it knows the syntactic category to which the corresponding syntactic component belongs. The denotation supplied as an argument is guaranteed to belong to this denotation class so long as the program syntax tree is syntactically correct. This knowledge permits the thunk corresponding to the denotation to be invoked directly with appropriate arguments, as described above in section 3.2.2.3.

It should be noted at this point that there is an alternative to the scheme of denotation class definitions just described. In this alternative scheme, each denotation is defined separately, as the sole member of a unique denotation type. Denotation classes are then specified as the union of all denotation types that correspond to the same syntactic category — the denotation class *expr*, for example, could be declared to be the

union of the types of the denotations for addition, constant, multiplication, and whatever other expressions are permitted. This scheme would permit the denotations for specific constructs to specify the number and denotation classes of the sub-component denotations to be supplied as arguments.

The disadvantage of the alternative scheme is that the denotation of a sub-component is no longer guaranteed to be of a particular procedure type, but can be any one of a union of types. This necessitates the use of a run-time tag check to determine the type of an argument before invoking the thunk.

The prototype implementation of EDS uses the scheme described originally.

Denotations differ from combinators in that they may contain references to the denotations of sub-components. These references have to be expanded into invocations of the appropriate "thunk" that has been bound as an argument. This expansion requires extraction of the corresponding denotation function, determination of the arguments that have already been bound for that function, and binding of the additional arguments now supplied. Some of the previously bound arguments may be thunks. In order to simplify this expansion, EDS requires that the denotation be referenced through a special *eval* construct that specifies the denotation class to which the denotation being invoked belongs.

3.2.2.7. The rest of the interpreter

Once all the combinators and denotations have been translated into Modula-2 and compiled, two things remain. First, we must generate the tree-builder that parses programs and constructs the tree of denotations that represents the executable version of the program, and secondly, we must provide a method of interpreting the denotation trees. Given the association between productions and denotations provided in an EDS

specification, the tree-builder is almost identical to an ordinary parser that constructs a parse tree of a program during parsing; it can be constructed from the syntactic specification of the language by a slightly modified parser generator.

Execution of the denotation tree is simple: The denotation tree is one large, deeply-nested function call, where some arguments (the denotations of subcomponents) are to be passed by name rather than evaluated before the call. The translated *eval* constructs will evaluate the thunks corresponding to these by-name arguments as required in the course of execution.

3.2.3. An example of an EDS specification

We will now describe the process of transforming a standard denotational semantics into an EDS-style specification, using as an example the language L introduced in chapter 2.

3.2.3.1. The semantic domains

3.2.3.1.1. Basic Values

As in section 2.2.2.1.1, the domain of basic values is

$$\mathbf{Val} = \mathbf{Int} + \mathbf{error}$$

The predefined domain **Int** corresponds directly to the machine representation of integers, which does not include a separate error element, and does not, in general, provide strict functions or report errors such as overflow. A more efficient, but less secure, implementation of L may elect to forego the inclusion of **error** in the domain of basic values; we include it here, in order to follow more closely the definition of L given previously.

3.2.3.1.2. Stores

We choose to use an opaque **Store** domain, implemented as a pointer to an area of memory set aside for this purpose. Actually, we will go further, and assume that the definition of L is *single-threaded* in the store, so that we can implement store updates and allocates by side effects on a single global store. David Schmidt recently developed sufficient criteria for single-threading [Schmidt85], but the current implementation of EDS does not verify that a semantics is single-threaded in a particular variable. EDS has been adapted to accept semantic definitions written in the more powerful continuation style (see section 3.3 below), and it is expected that most specifications will choose this style. As noted by Schmidt, semantics written in continuation style are trivially single-threaded in the store.

The opaque store domain does not affect other parts of the semantics, except that we can no longer apply the store directly to locations, but must instead use a **load** combinator to return the value stored at a particular location. This combinator, along with the other combinators that manipulate the store, will be described in the section on combinators below.

3.2.3.1.3. Environments

Although the store is conceptually a function domain, we chose to hide this in our implementation of it, and so did not have to contend with the fact that EDS does not have a facility for defining function domains. Our treatment of the environment domain in this section will show how this limitation can be overcome.

Environments are a rather special type of function, both in the way they are defined and in the way they are manipulated. An environment has the same value (*undefined*) for all but finitely many argument values. it can thus be modeled by a

default value together with a finite mapping. The choice of mapping depends on the ways in which environments are defined and used: a *hash table* will give easy updating and access, but make it inconvenient to save and restore environments, while an *association list* will slow down lookup somewhat but make it easier to save and restore environments.

Having chosen efficiency over generality in our definition of **Store**, we will make the opposite choice in defining **Env**; although the semantics of L (a language without block structure or scoping) do not require saving and restoring environments and would hence be most efficiently implemented by using a hash table for an environment, we choose the association list model for illustrative purposes. We will, however, manipulate the environment only through special combinators, thus hiding the implementation choice from the other components of the semantics, and easing later changes.

An association list is a list of pairs; each pair consists of an identifier and an environment value to be associated with the identifier. List domains can be created through use of domain definitions of the form

$$\mathbf{DLIST} = \mathbf{D} + (\mathbf{D} \times \mathbf{DLIST})$$

which defines **DLIST** to be a domain containing all finite sequences of elements of **D**. In this example, we will, for conciseness, assume that EDS is extended to provide a LISTOF domain constructor, together with CONS, CAR and CDR operations on elements of list domains. Using this extended notation, the environment-related domains for L can be defined as in Figure 3-2. Note that **Id** is a predefined domain, and **Loc** is an opaque domain that is implemented as a type "POINTER TO **Val**", where **Val** is the domain of storable values.

undefined= *undefinedval*;
EnvValue= undefined + Val + Loc;
Binding= Id × EnvValue
Env = LISTOF Binding

Figure 3-2: Environment domains for L in EDS.

3.2.3.2. Combinator definitions

The low-level combinators that manipulate stores and environments are the only sections of the semantics that need to know how these constructs are implemented. We will now define these combinators, using the implementations described above for the store and environment.

3.2.3.2.1. Store combinators

The **load**, **update**, and **allocate** combinators are used to manipulate the store. These combinators, as well as the combinator classes to which they belong, are defined in Figure 3-3. The domain **AllocRes** is defined to be (**Store** × **Address**). Note that the store is used only implicitly, since EDS uses the real machine store, with elements of **Loc** being pointers into actual memory. The parameter "s" is actually a dummy in this specification.

3.2.3.2.2. Environment combinators

Environment manipulations in L are simple: we can update the binding of a particular identifier, or we can look up the binding of an identifier. The implementation chosen is an association list of (**Id**, **EnvValue**) bindings. The value associated with an identifier can be updated by prepending the new binding to the association list, and

```
tload    = COMBCLASS( s : Store; a : Loc ) → Val;
tupdate  = COMBCLASS( s : Store; a : Loc; v : Val ) → Store;
tallocate= COMBCLASS( s : Store ) → AllocRes;

COMBINATOR load : tload;
var
        returnval : Val;
begin
        if a = nil then
                returnval ::= error( errorval );
        else
                returnval := a^;
        end;
        return(returnval);
end load;

COMBINATOR update : tupdate;
begin
        if a <> nil then
                a^ := v;
        return(s);
end update;

COMBINATOR allocate : tallocate;
var
        returnval : AllocRes;
begin
        returnval.Head := s;
        new(returnval.Tail);
        return(returnval);
end allocate;
```

Figure 3-3: Definition of Store combinators for L.

lookup must therefore return the first binding encountered, or *undefined* if none. Note that this implementation allows us to use pointers into the middle of the association list to access previous environments, thus eliminating the need to copy environments on every update, while not assuming single-threading of environments. Although we will not make use of this property here, specifications of languages that have several scopes would be able to use it to advantage. The combinators **define** and **lookup** are given in Figure 3-4.

```
tdefine  = COMBCLASS(e : Env; i : Id; v : EnvValue) → Env;
tlookup  = COMBCLASS(i : Id; e : Env) → EnvValue;

COMBINATOR define : tdefine;
var
        NewBinding : Binding;
begin
        NewBinding.Head := i;
        NewBinding.Tail := v;
        return(CONS(NewBinding, e));
end define;

COMBINATOR lookup : tlookup;
var
        returnval : EnvValue;
begin
        if e = nil then
                returnval ::= undefined(undefinedval);
        elsif EqualIde(CAR(e).Head, i) then
                returnval := CAR(e).Tail;
        else
                returnval := lookup(i, CDR(e));
        end;
        return(returnval);
end lookup;
```

Figure 3-4: Definition of Environment combinators for L.

3.2.3.3. The denotations and their classes

Each production in the syntax of the object language has a denotation associated with it. These denotations are divided into denotation *classes* which correspond to the syntactic categories in the object language, and to the denotational functions of the standard semantics of the object language.

3.2.3.3.1. Denotation classes

The denotation classes for L are defined in Figure 3-5. There is one denotation class corresponding to each of the semantic functions E, C, D, and P defined in section 2.2.2.2. **Res**, the domain of command results is defined, as before, to be **Store + error**. The domain **DRes** of declaration results is simplified to be just **Env + error**, in light of

```
E = DENOTECLASS(env : Env; s : Store) → Val;
C = DENOTECLASS(env : Env; s : Store) → Res;
D = DENOTECLASS(env : Env; s : Store) → DRes;
P = DENOTECLASS() → Val;
```

Figure 3-5: Denotation classes for L.

the single global store used; the store could be included in the result, but would needlessly complicate the example.

3.2.3.3.2. Denotations for expressions

There are three denotations belonging to the class *E*, corresponding to each of the productions for an <expr> in the syntax of L. These denotations are defined in Figure 3-6.

Some explanation is required here. The notation "SynArg$*n*" refers to the denotation of the n^{th} syntactic argument to this denotation. In cases where the argument belongs to a primitive domain (such as integers or identifiers), it is accessed directly; in cases where the argument is a (partially applied) member of a denotation class, the appropriate denotation is invoked via an (EDS-defined) macro which sets up the correct environment and arguments for the denotation (which has been passed by name, as a "thunk"). EDS defines one such macro for each denotation class; EvalE is the macro corresponding to the denotation class *E*.

3.2.3.3.3. Denotations for commands

The denotations for statements belong to the denotation class *C*. They are given in Figure 3-7.

```
DENOTATION NumberExp : E;
(* denotation of a numeral *)
var
        returnval : Val;
begin
        (* Convert numeral to integer, checking for overflow, etc. *)
        (* Ascii_to_Integer returns true if conversion successful  *)
        if Ascii_to_Integer(SynArg$1, i) then
                returnval ::= integer(i);
        else
                returnval ::= error(errorval);
        end;
        return(returnval);
end NumberExp;

DENOTATION IdExp : E;
(* denotation of an identifier *)
var
        idval : EnvValue;
        returnval : Val;
begin
        idval := lookup(SynArg$1, env);
        dcase idval of
                undefined :
                        returnval ::= error(errorval);
                Val :
                        returnval := idval::Val;
                Loc :
                        returnval := load(s, idval::Loc);
        end;
        return(returnval);
end IdExp;

DENOTATION AddExp : E;
(* denotation of a sum *)
begin
        return(plus(EvalE(SynArg$1(env, s)), EvalE(SynArg$2(env, s))));
        (* note that plus is strict, so need not check for error in sub-expressions *)
end AddExp;
```

Figure 3-6: Denotations for expressions in L.

```
DENOTATION NullCom : C;
(* denotation of a null command *)
var
        returnval : Res;
begin
        returnval ::= Res(s);
        return(returnval);
end NullCom;

DENOTATION SeqCom : C;
(* denotation of a sequence of commands *)
var
        returnval : Res;
begin
        returnval := EvalC(SynArg$1(env, s));
        if returnval ? error then
        (* no need to execute second statement, return error *)
                return(returnval);
        else
                returnval := EvalC(SynArg$2(env, returnval::Store));
        end;
        return(returnval);
end SeqCom;

DENOTATION AssignCom : C
(* denotation for assignment statement *)
var
        returnval : Res;
        expval : Val;
        idval  : EnvValue;
begin
        expval := EvalE(SynArg$1(env, s));
        if expval ? error then
                returnval ::= error(errorval);
        else
                idval := lookup(SynArg$1, env);
                if idval ? Loc then
                        returnval ::= Store(update(s, idval::Loc, expval));
                else
                        returnval ::= error(errorval);
                end;
        end;
        return(returnval);
end AssignCom;
```

Figure 3-7: Denotations for statements in L.

3.2.3.3.4. Denotations for declarations

The denotations for declarations in L are given in Figure 3-8. These denotations all belong to the denotation class *D*.

```
DENOTATION NullDec : D;(* denotation for null declaration -- no effect *)
var     returnval : DRes;
begin
        returnval ::= Env(env);
        return(returnval);
end NullDec;

DENOTATION SeqDec : D;(* denotation for two declarations in sequence *)
var     returnval : DRes;
begin
        returnval := EvalD(SynArg$1(env, s));
        if returnval ? error then
                return(returnval);
        else
                return(EvalD(SynArg$2(returnval::Env, s)));
        end;
end SeqDec;

DENOTATION ConstDec : D;(* denotation for constant declaration *)
var     returnval : DRes;
        expval : Val;
        envval : EnvValue;
begin
        if lookup(SynArg$1, env) ? undefined then
                expval := EvalE(SynArg$2, env, s);
                if expval ? error then
                        returnval ::= error(errorval);
                else
                        envval ::= integer(expval::integer);
                        returnval ::= Env(define(env, SynArg$1, envval));
                end;
        else
                returnval ::= error(errorval);
        end;
        return(returnval);
end ConstDec;

DENOTATION VarDec : D;(* denotation for variable declaration *)
var     returnval : DRes;
        envval : EnvValue;
begin
        if lookup(SynArg$1, env) ? undefined then
                envval ::= Loc(allocate(s).Tail);
                returnval ::= Env(define(env, SynArg$1, envval));
        else
                returnval ::= error(errorval);
        end;
        return(returnval);
end VarDec;
```

Figure 3-8: Denotations for declarations in L.

3.2.3.3.5. Denotation for a program

Finally, we can define the denotation for a program in Figure 3-9. The store s and environment env are assumed to be initially empty.

3.2.3.4. Translation Rules

Having defined the semantic domains, combinators, and denotations for L, all that remains is to specify the association between productions and denotations. Since we

```
DENOTATION Prog : P;
var
        s : Store;
        env : Env;
        val : Val;
        dres : DRes;
        cres : Res;
        idval: EnvValue;
begin
        dres := EvalD(SynArg$2, env, s);
        if dres ? error then
                val ::= error(errorval);
                return(val);
        else
                cres := EvalC(SynArg$3, dres::Env, s);
                if cres ? error then
                        val ::= error(errorval);
                        return(val);
                else
                        idval := lookup(SynArg$1, dres::Env);
                        dcase idval of
                                undefined:
                                        returnval ::= error(errorval);
                                Val:
                                        returnval := idval::Val;
                                Loc:
                                        returnval := load(s, idval::Loc);
                        end;
                end;
        end;
        return(returnval);
end Prog;
```

Figure 3-9: Denotation of a program in L.

have one denotation per production, and each denotation takes the denotations of *all* its syntactic components as arguments, this association is easily specified by including a denotation name with each production in the syntax of L. In the denotations given above, this association is given informally within comments (delimited by (* and *)).

3.2.3.5. Fixed points and iteration

EDS, as described above, has no fixed-point operator. The semantics of a **while** loop can be simulated in EDS, as given in Figure 3-10.

```
DENOTATION WhileCom : C;
var
        returnval : Res;
        expval : Val;(* Control expression value *)
        newstore : Store;(* Updated store *)
        cval : Res;(* Intermediate command result *)
begin
        expval := EvalE(SynArg$1(env, s));
        newstore := s;
        loop
                if expval ? error then
                        returnval ::= error(errorval);
                        return(returnval);
                elsif expval::integer <= 0 then
                        returnval ::= Store(newstore);
                        return(returnval);
                else
                        cres := EvalC(SynArg$2(env, newstore));
                        if cres ? error then
                                returnval ::= error(errorval);
                                return(returnval);
                        else
                                newstore := cres::Store;
                                expval := EvalE(SynArg$1(env, newstore));
                        end;
                end;
        end;
end WhileCom;
```

Figure 3-10: Semantics of iteration in EDS.

This definition actually corresponds fairly closely to the limit definition given in Figure 2-7. The extension of EDS to encompass semantic definitions written in *continuation* style, which is the subject of the next section, will make the fixed-point operation moot for iteration and other control-flow aspects of language specification.

3.3. EDS with continuations

Certain programming language constructs, particularly those that cause non-local transfers of control, are more cleanly described in a semantics using continuations, rather than the direct semantics we have used so far in EDS. Since most realistic programming languages contain such constructs, we would like to extend EDS to support semantic descriptions written using continuations. This section examines how the introduction of continuations affects EDS.

There are several things to consider when examining the impact of continuations on EDS. First and foremost, continuations are functions; we must determine whether the constructs provided in EDS for manipulating functions are sufficient to handle semantic descriptions written using continuations, and if not, how they must be extended to accommodate such definitions. Even if no changes are required, we have to examine how continuations affect the form of an EDS specification, specifically the denotation definitions and the translation rules. In addition, we must observe how continuations change the executable representation of a program, and how the translation and execution phases of the generated interpreter have to be modified to cope with these changes. As we shall see in chapter four, these changes are very significant in the integration of EDS into an IPE.

3.3.1. Domains, combinators, and denotations

The domain of continuations is a function domain; typically, a continuation maps a store into the language-specific domain of answers. Actually, the domain of answers is not used by most of the semantics; usually, a language definition contains only a few primitive continuations, which terminate program execution and extract an answer from the store. For example, a language may contain the primitive continuations **error** and **stop**, which represent an error exit and successful program termination, respectively. Either or both may extract some information from the program state at termination, and return this information (say by printing a message on the terminal) as the result of the program execution. Other continuations never manipulate values in the domain of answers; while they return values in this domain, they do so indirectly, by invoking some continuation on an appropriate store. Thus most continuations are defined by composing a store-transforming function with another continuation. In light of this characteristic, it is reasonable to use an opaque domain of answers, with primitive continuations such as **error** and **stop** defined as combinators that map stores into answers.

What of the other continuations, which are defined by composition? In order to handle them, we need a facility for defining function composition. We shall now extend EDS to allow a limited form of function domain; combinator and denotation classes will be considered to be domains. Although this extension allows the definition of function domains, the domains so defined have certain limitations. In particular, there is no facility for defining curried function domains; all function domains are defined to map a collection of arguments to a result in one step, with no intermediate results being created. In addition, the type checking requires name equivalence rather than structural equivalence. Thus, given the definition

$$\mathbf{C} = \text{COMBCLASS}(\, s : \mathbf{Store} \,) \rightarrow \mathbf{Ans},$$

the domains

$$\mathbf{Dom1} = \text{COMBCLASS}(\, c : \mathbf{C};\ s : \mathbf{Store} \,) \rightarrow \mathbf{Ans}$$

and

$$\mathbf{Dom2} = \text{COMBCLASS}(\, c : \mathbf{C} \,) \rightarrow \mathbf{C}$$

would not be considered equivalent.

Given this limited form of function domain, we can simulate the definition of a domain of continuations by using the notion of *completions*, introduced by Henson and Turner [Henson82] as the operational analogue of continuations.

Consider the language L defined using continuations as in Figure 2-8. Note that by right-canceling the store in the definition of *C*, the semantic function for statements, we can consider *C* to specify an equivalence between continuations. This equivalence can also be seen in the definition of the functionality of *C*. Since we have defined the domain of continuations to be

$$\mathbf{Cont} = \mathbf{Store} \rightarrow \mathbf{Ans}$$

the functionality of *C*, given originally as

$$C = \langle\text{stmt}\rangle \rightarrow \mathbf{Env} \rightarrow \mathbf{Cont} \rightarrow \mathbf{Store} \rightarrow \mathbf{Ans}$$

can be rewritten as:

$$C = \langle\text{stmt}\rangle \rightarrow \mathbf{Env} \rightarrow \mathbf{Cont} \rightarrow \mathbf{Cont}$$

The denotations of statements (with appropriate environments) can thus be considered to be *continuation transformers*. Hence the combination of a statement denotation, an environment, and a continuation, defines a continuation that is the result of applying the statement denotation to the environment and continuation in turn; this is really just another way of saying that the composition of a store-transforming function (a statement denotation with an environment) with a continuation yields a function in the

domain of continuations.

This recognition gives us the mechanism for defining the domain of continuations: First, define the domain of primitive continuations, which is simply the combinator class for the primitive continuations **error** and **stop**; second, note that a continuation can be either a primitive continuation, or a (statement denotation, environment, continuation) triple. The domain of continuations can thus be defined as the sum (disjoint union) of these two alternatives.

Invoking a continuation is no longer a simple matter of applying it to the store, however, since non-primitive continuations are now represented as closures. Invoking these closures requires calling the function representing the statement denotation, with the environment and continuation as arguments in addition to the store; the statement denotation performs the composition by calling the second continuation with the appropriate store. A new keyword, CONTINUE, has been added to EDS to simplify this task; the statement

CONTINUE contname(s);

will test the tag of contname and create an appropriate function call, with s as the store parameter. A CONTINUE statement should terminate the body of every statement denotation.

Other types of continuations, for example declaration continuations, can also be included in the completion framework. Assuming the definitions

$$\mathbf{DCont} = (\mathbf{Env} \times \mathbf{Store}) \rightarrow \mathbf{Ans}$$

and

$$D = \langle \text{decl} \rangle \rightarrow \mathbf{Env} \rightarrow \mathbf{DCont} \rightarrow \mathbf{Store} \rightarrow \mathbf{Ans}$$

note that they can be rewritten as

$$\mathbf{DCont} = \mathbf{Env} \rightarrow \mathbf{Store} \rightarrow \mathbf{Ans} = \mathbf{Env} \rightarrow \mathbf{Cont}$$

and

$$D = \langle\text{decl}\rangle \rightarrow \textbf{Env} \rightarrow \textbf{DCont} \rightarrow \textbf{Cont}.$$

We can then expand the completion sum domain to include the triple of a declaration denotation, an environment, and a declaration continuation. The body of a declaration denotation will bind its result environment with its supplied declaration continuation to form a continuation, and invoke the result through a CONTINUE statement.

3.3.2. Translating continuation semantics

The translation rules for a continuation semantics are more complicated than those of a direct semantics. In the case of a direct semantics, the translation rules were trivial — denotations were bound as arguments to other denotations in accordance with the syntax of the program. All denotations were defined identically, with the denotations of all syntactic subcomponents as arguments. In a continuation semantics, there is the problem of continuation arguments to contend with.

The most direct approach to dealing with continuations is to treat them as run-time arguments to denotations, just as we have been doing with environments and stores. However, note that the definition of the continuation of a particular statement in a program involves the continuation of the statement following it, and so on: continuations are defined *in reverse*, opposite to the flow of execution of the program.

Actually, this is not strictly true; There are really three components required to define a continuation (other than a primitive continuation): a statement denotation, an environment, and another continuation (we ignore for the moment declaration and expression continuations — they are analogous). Of these, statement denotations are always static, depending only on the structure of the program. The linkage that defines the composition of continuations is also a static property for most programming

language constructs — this claim will be substantiated later. Hence, if we can factor out the environment from continuations, we can define continuations statically, and bind them as arguments during translation.

The completion representation of continuations has already provided part of the mechanism we need to separate the environment part of a continuation definition from the (static) denotation part. Assuming we have a method of binding the denotation part of completions statically (this will be discussed in the next section), it remains to ensure that the appropriate environment is available and bound into the completions before they are invoked through a CONTINUE statement.

A simple way to accomplish this binding is to restructure our semantics. Instead of placing the environment before the continuation in the (curried) definition of the meaning functions, place it after; the meaning function for statements (for example) would then have functionality

$$C = \text{<stmt>} \rightarrow \textbf{Cont} \rightarrow \textbf{Env} \rightarrow \textbf{Store} \rightarrow \textbf{Ans}$$

We can then modify the domain of continuations to accept an environment argument in addition to a store, thus making the environment part of the run-time state of the program. This rewriting makes statement denotations (independently of environments) into continuation transformers, and permits the static determination of continuations.

This solution is satisfactory in the case where the environment is purely dynamic — that is, when all commands are executed in the environment that exists when they are invoked; it does not provide for execution in a previously bound environment, such as in the case of LISP "funargs," for example. In order to support other scoping and binding schemes, we must be able to create continuations that have environments bound in; we are thus forced to revert to our original scheme, with environments preceding continuations in the currying order. We can still simulate the scheme just

described, by binding the environment component of the completion just before invocation; in the case where the environment so bound is the current environment, we have dynamic scoping with shallow binding, as we would have with an environment that is part of the run-time state. We will leave unspecified for now the means of assigning environments under other scoping schemes.

We now return to consider the problem of determining the static component of continuations. Ravi Sethi [Sethi83] noted the correspondence between the continuation structure of a program and a flow-graph of the program. Consider Figure 3-11 (adapted from [Sethi83]), where each statement defines a continuation, in terms of a supplied continuation and the denotations of its subcomponents. Following Sethi, $statement represents the meaning of statement. Sethi considered states to be direct

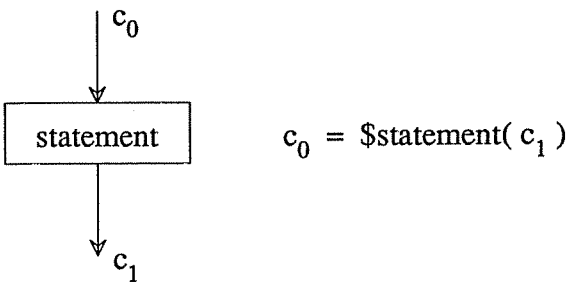


Figure 3-11: Statements as continuation transformers.

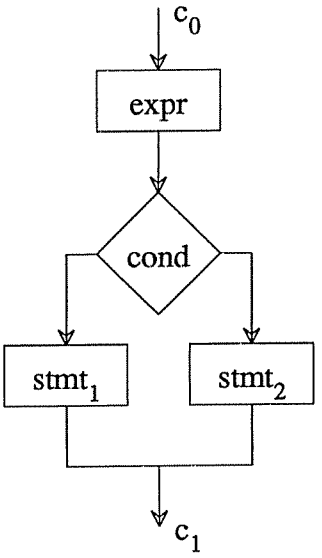
mappings from identifiers to values, thus sacrificing the advantages of a two-step (environment plus store) mapping, such as the ability to model aliasing. This simplification avoids the environment problem discussed above, and enables Sethi to treat statements as continuation transformers.

Sethi uses *Plumb*, a language based on the *pipe* notation for composing functions, to specify the construction of flow graphs for programs. With each production in the syntax of the language, Sethi associates a rule, written in Plumb, which describes the flow graph for that construct in terms of the flow graphs of its components and certain primitive combinators. The reader is referred to [Sethi83] for a full explanation; we present the technique here by means of some examples (Figure 3-12), adapted from [Sethi83]. For each construct, we give the syntax, the associated *Plumb* rules, and the resulting flow graph. The notation c_i refers to a continuation (corresponding to a point in the flow graph), and the boxes contain the flow graphs for the sub components that label them. The primitive combinator **cond** selects between one of its two continuation arguments based on the expression value supplied to it.

Note especially the flow graph and specification for the **while** loop. Continuations here are defined recursively (hence the keyword **cyclic** in the specification), and the corresponding flow graph contains a cycle.

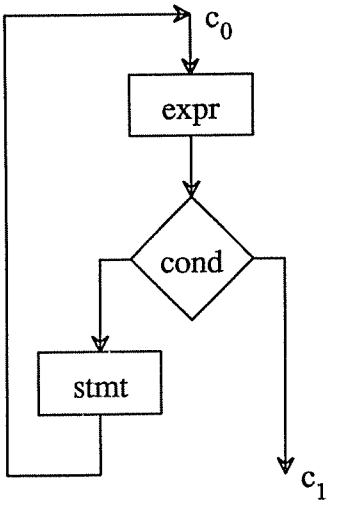
Using *Plumb*, Sethi is able to specify flow graphs for all the structured control constructs of the **C** programming language [Kernighan78].

In order to handle **gotos**, Sethi introduces an environment that maps identifiers (labels) into continuations (flow graph points). It should be noted that this environment may be kept completely separate from the environment used to determine the denotation of identifiers in object language programs; we shall refer to Sethi's environment as the *label* environment. Sethi notes that indirections through the label



<stmt> ::= IF <expr> THEN <$stmt_1$> ELSE <$stmt_2$>

{ \$\$ c_1 = \$expr | cond(\$$stmt_1$ c_1, \$$stmt_2$ c_1) }



<stmt> ::= WHILE <expr> DO <stmt>

{ \$\$ c_1 = **cyclic** c_0 = \$expr | cond(\$stmt c_0, c_1) }

Figure 3-12: Some examples of Plumb specification.

environment may be dispensed with once the flow graph for a program is constructed; this is a consequence of the static nature of labels in *C* — languages with label variables would require a run time label environment. For languages without label variables, however, our earlier claim that continuation structure is a static property is upheld.

Sethi goes on to describe how a *Plumb* specification can be combined with a grammar specification for a parser generator such as *yacc* to produce a translator that generates flow graphs for programs. His work can be easily adapted for our purposes; edges in his flow graph correspond exactly to the binding of completion arguments in EDS. We thus have a translation method that binds the static portion of completions for us, and we also see how the executable representation of a program changes now that we have continuations.

The introduction of continuations has changed the structure of the executable form of a program from a tree into a general graph. Loops and **goto**s form edges in the graph that do not correspond to the syntactic structure of the program.

3.3.3. Execution with continuations

Continuations make the flow of control in a program apparent; where the execution order of denotation functions in a program tree built with a direct semantics was determined implicitly by the order in which the denotation tree was traversed (which was decided by the order in which each denotation invoked the denotations of its subcomponents), the execution flow in a denotation graph built using continuations is explicit in the bindings of the continuations as arguments.

Previously, a denotation function would invoke each of its children in turn, and they would invoke their children, and so on; the execution state at any instant was

determined by the contents of the store and a stack of partially evaluated denotation functions representing the path from the currently executing function to the root of the denotation tree.

With the addition of continuations, each denotation function invokes precisely one other denotation (its continuation), and this invocation is always the last action before returning — continuation invocation is *tail recursive*. The execution state of a program denotation is thus determined by the denotation graph, the store contents, and a single current continuation value.

3.3.4. Procedure calls

Our above statement that a single continuation suffices to characterize the execution state of a program is not quite true in the presence of procedure calls. A procedure call involves two things: an environment switch (usually performed implicitly by the binding of an environment value to the procedure body when its denotation was created) and a saving of a return continuation. By analogy with actual implementations of languages, the saving and restoring of continuations during procedure call may be implemented by pushing and popping a stack of continuations.

The denotation of a **call** statement will push the continuation corresponding to the next statement onto the continuation stack, and transfer control to the continuation corresponding to the procedure being called; in the case of static scoping, this continuation may be part of the flow graph, but in the case of dynamic scoping, the continuation must be extracted from the environment. In the static case, the environment indirection may be eliminated as it was for labels.

A return from a procedure (either implicitly or through execution of a **return** statement), will simply pop the top continuation from the stack and invoke it.

3.4. Summary

This chapter has presented two versions of our approach to the generation of execution facilities from denotational semantics. The first version, for denotational semantics written in direct style, translates an object language program into a tree-structured denotation. A node in the tree is a pointer to a compiled representation of the denotation function of the appropriate construct, together with pointers to the nodes representing the denotations of the syntactic components of the construct.

EDS differs from most conventional semantics-directed compiler generators in using a (albeit extended) conventional programming language (Modula-2) rather than a special purpose functional language. As a consequence, EDS has to contend with the problems of defining and manipulating higher-order functions. We have described how the function manipulations encountered in denotational definitions can be simulated in EDS.

The second version of EDS incorporates the ability to handle semantic definitions written using continuations. Once again, the problems of function manipulation arise and are dealt with, by combining the techniques of *completions* [Henson82] and flow graph representation of continuations [Sethi83].

Although the issue has never been explicitly raised in this chapter, EDS was designed to complement a language-based editor generator, thus providing the capability to generate a *complete* integrated programming environment. The following chapter will describe how the design choices made in EDS enable incremental translation and interactive execution to be achieved with reasonable performance, thus fulfilling the goal of providing a truly integrated programming environment generated completely from specifications.

Chapter 4

EDS for an Integrated Programming Environment

We have thus far discussed EDS as a stand-alone interpreter generator. We will now consider how EDS can be combined with a generator of language-based editors to form an IPE generator.

As a first step, let us examine how an interpreter generated by EDS can be incorporated into a language-based editor to produce an IPE; if this integration is straightforward, we have an IPE generator, albeit one with two quite distinct components — EDS and the LBE generator. Subsequently, we can discuss how EDS and an LBE generator may be more completely integrated, and how such integration may be exploited to improve the generated IPE.

There are several LBE generators described in the literature, based on a variety of paradigms, and using different specification techniques. Rather than selecting a specific LBE generator from among the many candidates, we will describe the integration in terms of a generic LBE that has much in common with LBEs generated by a variety of generators, with reference to specific systems where appropriate.

In describing how an EDS-generated interpreter interfaces with an LBE, there are two aspects to consider — translation and execution. Chapter 3 has already described how an EDS-generated interpreter translates a source language program into its denotation; what we must examine here is how this translation can best be performed in the context of an IPE, keeping in mind the aim of reducing the length of the *edit-compile-debug* cycle as much as possible.

In the case of execution, the special nature of an IPE once again demands something more than a typical stand-alone interpreter; *interactive* execution is the key concept. We will describe several features of previous IPEs, and show how they may be incorporated into an EDS-generated interpreter.

4.1. A generic LBE

Many LBEs have been described in the literature, and many LBE generators as well. We will attempt here to describe a "generic" LBE that is in some sense a least common denominator of the various approaches that have been adopted in the past. We do not claim that this LBE is somehow the "best," or even as good as any of the others — in fact, the aim is to describe a "least" LBE that can be mutated fairly easily to fit any of the popular paradigms for LBEs. The description of this generic LBE will, of necessity, be incomplete; in particular, aspects of the LBE that have no direct bearing on the interface with EDS will be left unspecified.

A language-based editor usually represents programs internally as something other than simple text. Although the exact representation may take many forms, the aim is to facilitate structured manipulations of the program, which usually implies knowledge of the syntactic structure of programs. It is thus reasonable to assume that an LBE always either has, or can easily simulate, access to a (abstract or concrete) syntax tree of the program being edited. For simplicity, we will assume that our generic LBE stores programs internally as abstract syntax trees, with editing operations being either specified as, or converted into, tree manipulations.

Our LBE must have a way of displaying programs on screen in (more or less) conventional textual form in order to permit user interaction. We shall assume that this is done by creating a screen image from the internal representation, and that there is a

relatively simple method of accessing the screen image corresponding to a particular node in the syntax tree.

As we describe the process by which an EDS-generated interpreter is integrated into an LBE, we will come upon other facilities that are required in the LBE. Rather than describe these facilities now, we shall extend our generic LBE as the need arises. In some cases this extension will take the form of adding a particular feature not found in current LBEs, in other cases we will describe how the needed feature may be simulated with the facilities already common in LBEs; in the event that there are competing schools of thought on the way certain features are implemented in LBEs, we will attempt to outline the implementation of the required feature within the framework of each of the paradigms.

4.2. Translation

One way to effect the translation of a program into its denotation is through the use of a conventional stand-alone EDS-generated interpreter. This would involve generating a textual representation of the program, writing it out to a file, and invoking the EDS-generated interpreter on this file. The interpreter would then scan and parse the program contained therein, and convert it into a tree-structured denotation as described in chapter 3.

Quite apart from the inefficiency involved in converting a syntax tree representation of a program into a textual one, which is then immediately parsed again by EDS, there is a philosophical objection to the translation method just described. The aim of integrating tools into a common environment is to shorten the *edit-compile-debug* cycle as much as possible. Permitting rapid transitions between editing and execution is one way to further this aim. We will carry this approach to its extreme — we would

like to provide *instantaneous* transition from editing to execution, or at least reduce the transition time to the point where the user is unaware of a distinct translation phase, but rather imagines the program to be directly executable at all times.

In order to sustain this illusion, we must be able to maintain the executable representation of the program *incrementally* during editing. At the same time, this incremental processing should not significantly degrade the perceived response of the LBE to the user's editing commands.

A denotational semantics has two properties that are extremely useful for the purpose of incremental translation. Firstly, a denotational semantics is *syntax-directed* — denotations are specified for constructs based on their syntactic structure. A particular denotation depends only on the denotations of its immediate constituents, which are its direct children in the syntax tree. Secondly, the semantics is *modular* — the denotation of a construct depends on the denotations of its components, but the extent of this dependence is strictly delimited. For example, the denotation of an assignment statement includes the denotation of an expression; if the syntax tree is altered by replacing this expression, its denotation can be replaced by that of the replacement expression without requiring any further alterations to the statement denotation.

In the case of EDS, where denotations are represented as trees with pointers to the denotations of sub-components, this replacement is a simple matter of updating the argument pointer for the appropriate component of the statement denotation to point to the new expression denotation. The actual code for denotations, as was noted in chapter 3, is compiled into machine code at interpreter generation time, and it is merely pointers to the denotations that are manipulated during construction and modification of the executable representation. For example, Figure 4-1 shows the changes necessary when the statement

```
foo := 1;
```

is changed to

```
foo := foo + 1;
```

In the denotation for the assignment statement, the pointer to the denotation of its second sub-component (the expression whose value is to be assigned) must be changed from its original value (pointing to the denotation of the constant expression "1") to point to the denotation of the new expression (the sum "foo + 1"). The solid lines indicate the original pointer values, and the dashed line indicates the updated value.

Thus, for a direct semantics, we see that maintaining the executable representation of a program is analogous to maintaining the syntax tree of the program, which the LBE presumably already does. If EDS were to be integrated with the LBE generator,

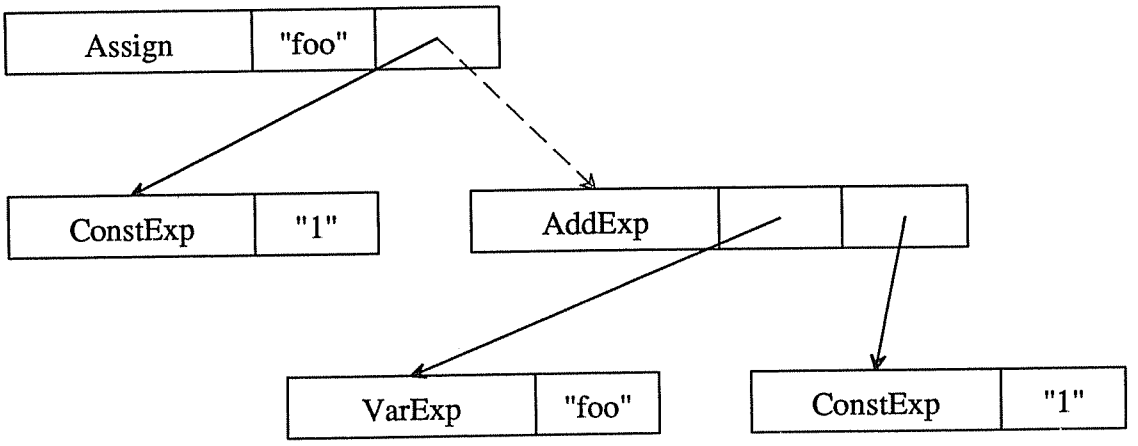


Figure 4-1: Incremental updating of denotation pointers.

the necessary denotation pointers could be incorporated into the definition of the tree nodes used by the LBE.

For an EDS interpreter added as an afterthought, we need a way of maintaining the denotation tree in parallel with updates to the syntax tree. Fortunately, although LBE generators do not agree on the way they would do so, they all provide a reasonably easy way to maintain the denotation tree.

Since the denotation tree of a program depends only on the structure of the program syntax tree, any LBE that provides support for static semantics must theoretically be capable of maintaining the denotation tree. In fact, in the case of the two most common approaches to static semantics in LBEs, action routines and attribute grammars, the denotation tree can be maintained with minimal effect on editing response, since the extent of the change to the syntax tree delimits the extent of the changes to be made to the denotation tree. The amount of work that has to be done for each syntax tree modification is thus bounded by a constant.

In the case of a continuation semantics, the situation is somewhat more complicated. Although the modularity of the definition is preserved and the executable representation can be constructed purely through pointer manipulations, there is the problem of continuation arguments to contend with. Continuation links are troublesome for two reasons. Firstly, they do not always point to nodes that are immediately adjacent in the syntax tree, as do the pointers to sub-component denotations in a direct semantics. Secondly, continuation links may form loops. In particular, the denotation of a program containing a **while** loop or backward jumping **goto** statement will contain continuation links that form a loop.

The non-local nature of continuation links has the potential to degrade seriously the editing response of an LBE that updates them incrementally, since the scope of

changes to the denotation graph is no longer confined to the production being modified. Thus although the work to be done is still constant for each editing operation, the nodes involved may be arbitrarily far apart; in the case of an LBE based on an incremental attribute grammar evaluator, this could mean examining all the intervening nodes in order to propagate the changes. In practice, however, the effect is minimal, for two reasons. Firstly, the frequency and extent of these non-local updates is limited; in most cases, the flow graph of a program corresponds very closely to the syntax tree, deviating only in the case of non-sequential transfers of control, such as **gotos** and loops. In a typical program, the percentage of such nodes is modest. Secondly, the problem of non-local updating arises in other contexts within an LBE, such as in associating identifier uses with their corresponding declarations, and has been recognized as a problem, particularly in the case of attribute grammar based editors. As a result, techniques have been developed to minimize the impact of such updates on editing response [Reps86].

The existence of loops in the flow graph of a program must be noted when specifying a flow graph constructor using an attribute grammar. Circular dependencies among attributes in a tree can cause the attribute values to be undefined, and attribute grammar evaluators will usually reject a grammar that (even potentially) permits circular dependencies. Once again, the problem has been noted and a solution advanced by the designers of LBE generators based on attribute grammars. The Synthesizer Generator [Reps84] permits the definition of *local* attributes that are associated with a *production* rather than a grammar symbol, and also allows an attribute to be a *pointer to* another attribute. The problem of circular continuation dependencies can then be removed by associating an *entry* attribute with each node that has a denotation, and setting the value of *entry* to be a pointer to the denotation. Other denotations that refer to

this one as a continuation will instead refer to the *entry* attribute. Since the *entry* attribute value depends only on the *location* of the denotation, and not its value, the circularity is broken. This strategy for dealing with circularities was previously described in [Mughal85], which used a flow graph approach to generating run-time facilities in association with the Synthesizer Generator. Using these *entry* attributes, the flow graph of a **while** statement as given in figure 3-12 would be modified as in figure 4-2.

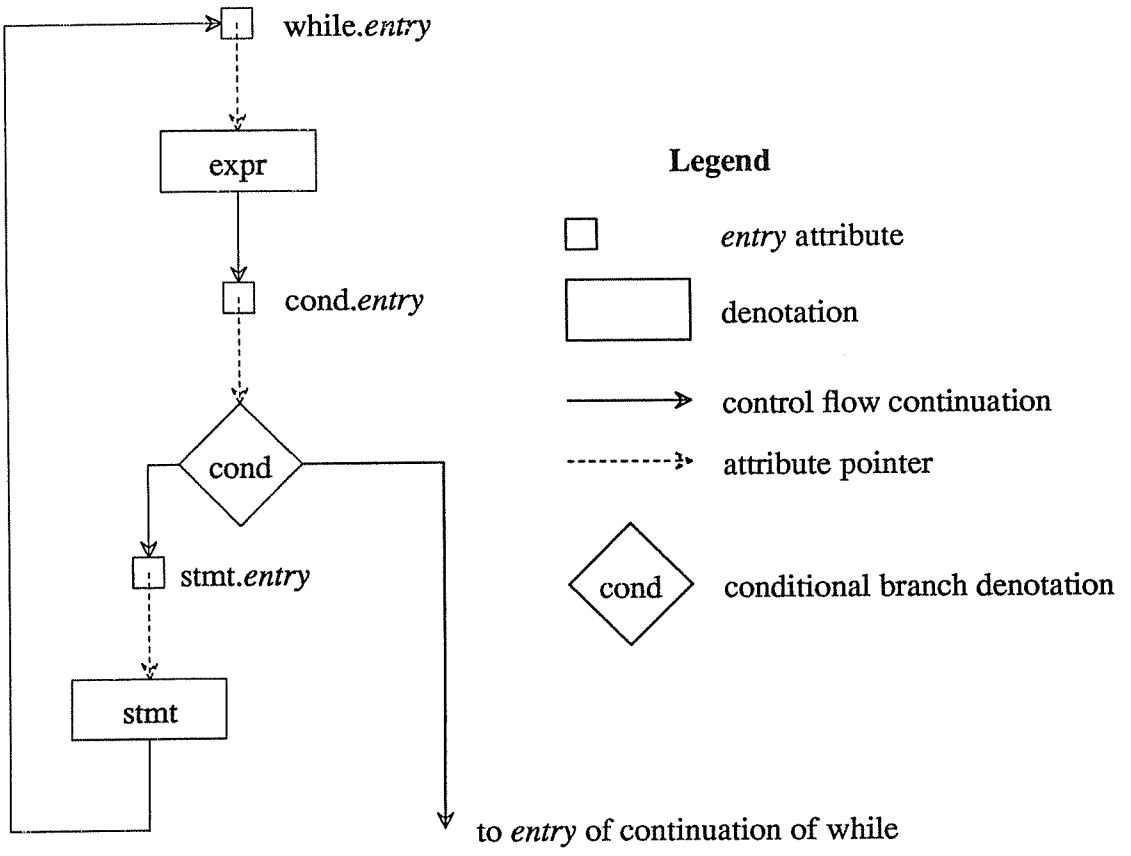


Figure 4-2: Flow graph for while loop with entry attributes.

Note that there is no loop comprosed entirely of solid arrows — the dotted attribute pointers serve to break the circularity.

4.3. Execution

Once we have a denotation tree for a program, we can interpret it on command as described in chapter 3. However, we would like to go beyond the facilities provided by a simple interpreter and utilize the special nature of an integrated environment to provide features that speed up the debugging process, thus shortening the development cycle. Choosing and evaluating the features that achieve this goal is beyond the scope of this thesis; instead, we will draw on previous work, selecting illustrative features from hand-coded IPEs and illustrating how they may be incorporated into the execution facilities generated by EDS. Although the features described originated in many different systems, they were all present in the Cornell Program Synthesizer [Teitelbaum81], which we will adopt as our benchmark.

The goal we wish to achieve is *interactive execution*; the user should be kept aware of the progress of execution, both in terms of the locus of execution in the program and the run-time state. Furthermore, the user should be able to suspend execution at any time, and interact with the program's run-time state; although we may wish to limit this interaction to be read-only, there are cases when it may be reasonable to allow the user to modify the run-time state and then resume execution. Carrying this idea even further, we can consider the consequences of permitting the user to modify the *program* itself before resuming execution.

In discussing the execution aspects of an EDS-generated interpreter integrated into an IPE, we will assume a continuation semantics. There are two reasons for this assumption: firstly, continuation semantics are, of necessity, more common in the

specification of actual programming languages, and secondly, the use of continuations makes program flow explicit, aiding the implementation of tracing and debugging facilities. In light of these advantages, we envisage the use of continuation semantics exclusively in actual specifications.

4.3.1. Flow Tracing

The high bandwidth of modern video display devices permits the flow of control to be traced on the screen during execution. The Cornell Program Synthesizer, for example, used the screen cursor to indicate the location of the instruction pointer at each moment, redrawing the screen as necessary whenever control passed outside the display window.

This feature can easily be incorporated into an interpreter generated using EDS. The most natural place to perform the display update necessary to switch the highlighting is during invocation of a continuation. Each continuation includes the denotation of a particular syntactic entity that will be the locus of execution when the continuation is invoked; we can modify the effect of a CONTINUE statement in the EDS specification language to test a global flag and, if set, highlight the screen image of the syntax tree node whose denotation forms the continuation being invoked.

In order to do so, two things are necessary — we must be able to find the tree node associated with a particular denotation, and we must be able to find the screen image for that tree node. If flow tracing is not to slow program execution unacceptably, these operations, as well as the actual highlighting itself, must be reasonably efficient.

Finding the syntax tree node corresponding to a denotation is easy to do — as noted in the discussion on translation, there is a direct correspondence between tree

nodes and denotations. If the denotations are included as a field in the tree nodes, as might be the case in a fully integrated generator system, the correspondence is trivially apparent; if the denotation tree is maintained separately, we would require an additional tree node pointer in each denotation, in addition to the pointers to denotation functions and sub-components, but this is a relatively minor overhead. In the case of the Synthesizer Generator, where denotations could be maintained as attributes, there is a simple mapping from attributes to the node with which they are associated.

Mapping syntax tree nodes to screen images, and highlighting the screen image, are functions that an LBE must already include. However, flow tracing, where screen updating frequency is controlled by execution speed, may demand higher performance than editing, where screen updating frequency is dependent on the rate at which the user initiates program changes.

One complication that has not been mentioned thus far is *un-highlighting* the screen image of a node when control leaves its denotation. Once again, the logical place to accomplish this task is during the transition from one continuation to another, namely, the execution of a CONTINUE statement. The problem that arises is determining the denotation that is currently being executed. Recall that the code for a denotation function may be shared among many denotations (such as the denotation function for an addition expression being shared among all addition expressions in the program), and hence there is no unique mapping from denotation functions to denotations. One possible solution is to pass the denotation itself as a parameter to the denotation function upon invocation, but there is a better way — since there can be only one denotation active at any time, it suffices to have some way to identify the *current denotation*. This current denotation may be a global location, corresponding roughly to an *instruction pointer*. CONTINUE statements would thus (assuming flow tracing is in

effect) perform the following sequence of actions:

- Un-highlight the current denotation.
- Extract the new current denotation from the continuation being invoked and install it as the current denotation.
- Highlight the screen image of the syntax tree node corresponding to the current denotation.
- Invoke the continuation as in a stand-alone interpreter.

4.3.2. Providing an execution monitor

Several interactive execution features, including breakpoints, single-stepping, and slowing down execution during tracing to follow the execution flow more easily, require the ability to, at various points during execution, transfer control outside the program, perform some action, and perhaps resume program execution.

As was the case with flow tracing, continuation invocation is the logical place to interpose these actions. We will now describe an efficient way of optionally inserting a (possibly empty) set of actions in the continuation invocation. Since the denotations are compiled at interpreter generation time, whereas the determination of the actions to be interposed is made just before execution, this insertion cannot be done through the use of compile-time flags on the denotation functions.

Consider the implementation of the CONTINUE statement. Conceptually, it consists of extracting a denotation function pointer and its arguments from a continuation, adding a store argument, and calling the denotation function with appropriate arguments. However, this call is the last action performed by the calling denotation function, and hence is tail recursive. Furthermore, the nature of a continuation semantics is such that only one denotation function will ever be active at a time. We can therefore optimize the continuation invocation as follows: first, replace the run-time argument

stack by a fixed argument area which is overwritten with the arguments for each new call; second, noting that we have already "pushed" the arguments by installing them in the argument area, and that the tail recursion obviates the need to save the return address, replace the call by a jump to the code for the denotation function.

This jump is now our foot in the door of program execution. By inserting a no-op prior to the jump, and replacing the no-op with a co-routine call to an execution monitor when desired, we can provide support for interactive execution with minimal overhead (execution of a no-op per continuation invocation) during normal execution.

Conceptually, we can look at the (possibly empty) call to the execution monitor as an extra store-transforming function that will be composed with every continuation. The return address saved when the monitor is called, together with the contents of the store, the argument area, and the procedure call continuation stack, comprise the entire state of the suspended execution, and so define a continuation that may be invoked to resume execution. If the execution state is unaltered, simply transferring control to the jump (returning from the monitor) will suffice to resume execution.

The execution monitor (or monitors — we can have several different monitors that may be interposed in a continuation invocation) has complete access to the state of the suspended execution, as well as freedom to interact with the rest of the IPE. In addition, the ability to interpose monitors selectively gives us tremendous power. For example, the user may interact with the monitor to query the current value of a variable, which can be determined from the execution state.

As a further example, suppose we wish to "watch" a variable, and update a screen display whenever its value changes. This can be done by interposing a display routine before every continuation that performs an update on the variable value.

Consider also uninitialized variables. Combinators that access the store can be modified to query the user for a value whenever they would otherwise return an unitialized or error value, assuming the store implementation permits the detection of such values. In the case of simple variables, the value may be input directly, whereas structured variables could use a structure editor to ensure the correctness of the value supplied. The domain specification for the return value would provide a specification for the structure editor invoked, using the techniques for editor generation from type declarations described in [Dewan86]

To make the most effective use of this ability to interpose actions in the execution path, we require the ability to identify sets of continuations (flow graph nodes) that satisfy certain criteria. In the case of watching variable "foo," for example, we need to identify all nodes with denotation function "Assign" and variable "foo." The incorporation of relational databases into IPEs, as proposed in [Linton84] and further developed in [Horwitz85], will greatly simplify this task. Assuming the existence of a relational database containing suitable information, many interactive execution and debugging features can be characterized by specifying a query that yields a set of flow graph nodes and a monitor continuation to be interposed before those nodes.

4.3.3. Incomplete programs

Now consider the execution of an incomplete program. Let us define a *default* denotation for each syntactic class. This default denotation will be inserted into the flow graph as the denotation of any missing or unexpanded construct. Consider what we could do when this default denotation is invoked at run-time. One option would be to suspend execution and force the user to supply the missing program fragment; in an editor generated by the Synthesizer Generator, for example, the denotation might open a new buffer with a root of the appropriate syntactic class, and graft the supplied sub-

tree onto the main program tree.

Another alternative would be to permit the user to simulate the effect of the unexpanded construct; the denotational nature of the semantics ensures that this is acceptable provided the default denotation enforces appropriate constraints on the user-supplied result. Thus the default denotation for an expression might request an integer value from the user, and the default denotation for a statement might permit the user to modify the store, since statements transform stores.

4.3.4. Interleaving editing and execution

The discussion of incomplete programs above introduced the concept of resuming execution after modifying a program, although the modification permitted was limited to the expansion of a previously unexpanded (and unexecuted) construct. Can we extend this concept to permit resumption of execution after other forms of modification?

Before we consider the question of implementing interleaved modification and execution, we must clarify the concepts involved. Let us introduce some terminology. We will use the term *original program* to refer to the program that was executing before suspension, and the term *modified program* to refer to the program after modification. The steps involved in interposing a program alteration in an execution are thus:

- Interruption of the executing original program.
- Transformation of the original program into the modified program.
- Resumption of the modified program execution.

Clearly, it is always possible to obtain some result when resuming execution after program modification, but for this result to be useful it must bear some relationship to

the correct one. A strict definition of correctness would require that the results obtained when resuming execution of a program be identical to those obtained if the modified program were executed from the beginning, and of course this result could be trivially obtained by restarting execution from the beginning of the (modified) program. We would like, however, to obtain the same effect without having to start over.

What we need then, is a characterization of the *state* of a suspended execution, and a way of modifying this state in accordance with modification of the program. Let us first characterize the state of a program execution; we will do so in terms of the internal representation of the program and the various denotational semantics constructs that accompany it, in order to maintain language independence.

We include the executable representation of the program (a denotation tree or flow graph) as part of the state of the program execution; in terms of the denotational semantics of the language, the denotation of a program is considered to be part of its execution state. The discussion of incremental translation earlier in the thesis has already dealt with the transformation of this component of the execution state in response to changes in the program.

Another obvious component of the execution state is the current locus of execution in the program — the "program counter." In the case of a direct semantics, the execution locus is defined by the current denotation together with its ancestors; in the case of a continuation semantics, the execution locus is defined by the current continuation together with the stack of continuations saved as return addresses from procedure calls.

The current store and environment are obviously components of the execution state of a program; although their exact nature will vary from language to language, they are language-independent concepts common to most denotational definitions.

Finally, any other global variables in the denotational specification, such as an expression evaluation stack, or a procedure call environment stack, are also components of the execution state of a program.

Having identified the elements that determine a program's execution state, we turn now to ways of mapping the execution state of a suspended program in accordance with modifications to the program itself.

As mentioned previously, the incremental translation techniques already described suffice to transform the denotation of the original program into that of the modified program; it remains to determine changes necessary to other components of the execution state to reflect the program modification.

It would appear at this point that we must relax our requirement of absolute correctness; it does not seem feasible to modify the execution state to be consistent with the state that would be obtained if the modified program were started anew. For example, the modified program, if executed, may never reach the point at which the original program execution was suspended; furthermore, this reachability question is known to be undecidable.

If we abandon absolute correctness, what then shall we require in its stead? A reasonable choice seems to be to require that all aspects of the execution state that are statically determined be correct, but permit dynamic aspects (such as the contents of the store) to have incorrect values; where we allow (possibly) incorrect values, it seems reasonable to retain the current values.

In addition to this weakened form of correctness, we would like to impose another condition — safety. That is, resumption of execution should not result in any error that could not have resulted from a normal program execution. Note that we are not proscribing errors that would not occur during execution of the modified program

from the beginning, which would require strict correctness in many cases; we are merely forbidding the introduction of a new class of errors. Thus uninitialized variables or zero-divide errors will be tolerated, even if they would not have occurred if the modified program were executed from the beginning, but faults in the program interpretation mechanism will not — situations that may precipitate such faults should be noted and appropriate actions taken before resuming execution.

It may be argued that we should ensure that no spurious errors are introduced as a result of resuming execution, but we feel that such a requirement is excessively restrictive, being equivalent to a requirement of absolute correctness. We feel that, to a competent programmer who understands the ramifications of resuming execution and uses this capability with care, the convenience of resuming execution outweighs the disadvantage of an occasional spurious error; we will leave it to the discretion of the user of an environment to attempt resumption only in cases where such resumption may reasonably be expected to produce meaningful results.

What then are the situations where catastrophic failure may result, and how can they be detected? One class of such situations involves cases where elements of the execution state that control further execution are no longer consistent with the modified program; for example, deletion of the statements corresponding to currently active denotations would be such a case. For a direct semantics, there is one currently executing denotation in a suspended execution, but in addition, the ancestors of this denotation in the denotation tree also serve to define the current locus of execution. In the event that any of these denotations is altered, it is unsafe to resume execution directly. Similarly, for a continuation semantics, the current continuation stack (resulting from procedure calls) serves to determine the execution locus. Any program modifications that invalidate the continuation stack (including the current continuation)

preclude direct resumption.

In such a case, what are the alternatives? The simplest choice is obviously to restart execution, but in most cases we can compromise; some integrated programming environments periodically save state as a way of supporting simulated *reverse execution*. Such a facility could be included in an EDS-generated interpreter, as we have already identified the components that determine an execution state; an actual implementation may be able to optimize such checkpointing by only saving part of the state, determined by change and relevance.

If the execution state has been checkpointed, we can "roll back" execution to a point preceding the invocation of the changed denotation, and then resume execution from that intermediate point; in the case of a direct semantics, this corresponds to moving up the denotation tree above the point of change, and then resuming; in the case of a continuation semantics, we must roll back from procedure calls until all invalidated continuations have been removed from the stack.

If expression semantics have also been written in continuation style, using an expression stack, and if execution can be suspended during expression evaluation, it is possible that modifications to the expression under evaluation could invalidate the stack, and more importantly, make it unsafe to resume execution, by creating a potential stack underflow situation. In such cases, we would have to roll back execution until the stack is empty, and re-start expression evaluation; if expression evaluation is side-effect free, this rollback is relatively inexpensive.

There remains one component of the run time state that we have not considered — the environment. Since the environment holds the key to interpretation of the store, we must be able to maintain the consistency of the environment during program modification if resumption is to be meaningful. There may be more than one

environment encoded in a suspended execution state; in particular, procedure returns will usually require restoration of a previous environment; such environments will be saved along with the return continuation when the procedure is called.

The simplest approach to maintaining environment consistency is to forbid resumption whenever an existing (or saved) environment is disturbed; we could of course use the rollback technique employed for procedure calls here as well. It is tempting, however, to try to transmute extant environments when programs change. A full analysis of this topic is beyond the scope of the present work, but we present some ideas that might merit further study.

Consider additions or deletions of variable declarations. For the sake of argument, we will assume that the resulting program does not contain undeclared variables — the detection of undeclared variables, though necessary, is inherent in semantic specification and not an artifact of resuming execution after modification. We will also restrict these additions and deletions to leave unaffected the remainder of the environment; for example, they should not (through the scoping mechanism of the language) cause other variable occurrences to be re-bound (in the modified program) to a definition other than the one to which they were bound in the original program. If the environment modifications satisfy these criteria, then it is clear that a suitable implementation of environments will permit replacing old environments with new without affecting correctness of the execution state; all that is required is that the new environment be identical to the old in the intersecting portion, specifically allocation of locations (values must be identical in any case).

Changes to declarations are more troublesome — since changes to a type or variable declaration may require that the space allocated for a variable be enlarged, or the contents of the store be re-interpreted. It is not clear whether it will in general be

worthwhile to do so; it may be simpler to roll back execution to a point where none of the environments have been modified.

Changes in procedure parameters are an especially troublesome form of environment modification; it would appear that in such cases rollback is the best solution.

4.4. Incorporating compiled code

In order to achieve greater execution speed, some interpretive systems allow parts of a program to be compiled, and mix this compiled code in with interpretation of the remainder of the program.

Interposing a section of compiled code in a denotation tree or a flow graph is simple — the code segment may be packaged as a denotation function with the appropriate parameters and return type and inserted into the denotation of the program just like any other denotation. What is not so simple is the interaction between the compiled and interpreted parts of the program.

For this mixed-code strategy to succeed, the compiled version of the code must be able to mimic exactly the behavior of the equivalent interpreted version. We determine the behavior of a program fragment by its effect on the execution state of the program; the components of this execution state were described previously in the section on interleaved editing and execution.

Since EDS-generated interpreters use a model of the store that closely fits the typical compiled model, and access and update this store through predefined combinators, it is easy to ensure that compiled versions of program fragments respect this store model. Similarly, the run-time expression stack can easily be made compatible with both interpreted and compiled code.

Procedure calls involving compiled code require a little care; there are three cases to consider:

(1) Compiled procedure called from interpreted code. In this case, a simple interface denotation can be wrapped around the compiled procedure to handle the call and return in the way that the interpreted code expects.

(2) Compiled procedure called from compiled procedure. In this case, there is no interface necessary, and the call can proceed as a normal call would in compiled code.

(3) Interpreted procedure called from compiled procedure. This case is more troublesome, especially if we do not know at compile time whether the called procedure will be compiled or interpreted. One possible solution is to assume that all procedures called from compiled code will be compiled, and compile a special stub that reverts to interpretation if such is not the case.

In the above discussion, we have ignored environments. One of the principal sources of inefficiency in an EDS-generated interpreter is the indirection through the environment for identifier references; we would hope that compiled code can avoid this inefficiency. The only way to ensure this is to "freeze" that portion of the environment that is referenced by a compiled code segment when the segment is compiled; subsequent modifications to the environment will force re-compilation of the fragment. Note that this dependence on the environment ties in with our previous experience in studying interleaved execution and modification — in both cases additions to or deletions from the environment that leave unchanged the "interesting" part of the environment are harmless, provided environments are implemented appropriately.

In summary, then, compiled code appears to be feasible provided the interactions between the environment of the compiled code and that of the interpreted portion of

the program are minimized. Thus it makes sense to compile program sections at a granularity of modules, where environment interactions are limited to parameters and global variables (which can be allocated in the (temporal) order of their definition, thus minimizing environment-caused recompilation).

The problem of minimizing module recompilation as a result of environment changes has been studied by Walter Tichy [Tichy86]. We expect that work in this area will be applicable to both the execution resumption and mixed-code problems in integrated programming environments, as a result of the shared aim of maintaining consistency between similar environments.

4.5. Summary

In this chapter, we have shown how EDS, and interpreters generated by it, may be integrated with an LBE generator, and LBEs generated thereby, to produce a generator for, and examples of, complete IPEs. We have aimed for two principal capabilities in the generated IPE: incremental translation and interactive execution.

The paradigm of instant executability at all times, which we feel is an integral part of a "true" IPE, led to the requirement for incremental translation. We have shown how the tree-structured denotations of EDS-generated implementations, as well as the flow-graph executable representations used in the case of continuation semantics, can be easily created and maintained incrementally using facilities found in current LBEs.

On the subject of interactive execution, we have shown how many of the features of hand-coded IPEs such as the Cornell Program Synthesizer can be incorporated into interpreters generated from continuation semantics by EDS. The provision of "hooks" which facilitate the insertion of arbitrary monitor functions into the continuation invo-

cations of an executable representation allow the implementation of many interactive debugging features, while minimizing overhead during un-monitored execution.

We have thus shown the feasibility of generating execution facilities for IPEs from denotational semantic specifications, while retaining the interactive features that made hand-coded IPEs so attractive.

Chapter 5

Conclusions

5.1. Summary of work

Integrated programming environments that support program execution are a natural extension of language-based editors. Just as was the case with LBEs, there is a strong incentive to automate the generation of IPEs, which requires a formal specification method for all the language-dependent aspects of the IPE, namely *syntax*, *static semantics*, and *dynamic semantics*. Previous work on LBE generators has addressed the issues of syntax and static semantics, but attempts to incorporate dynamic semantics have been less successful.

We have presented an approach to the problem of generating execution facilities for IPEs that is based on denotational semantics. A denotational semantics for a language provides a modular, syntax-directed mapping from programs in the language into mathematical functions, and can be used as a specification for an implementation of the language. Denotational semantics has been the basis for much work in the area of semantics-directed compiler generation.

In adapting compiler-generation techniques to IPEs, there were three issues to be addressed:

- The severe *performance problems* normally associated with compiler generators based on denotational semantics.
- Support for *incremental translation*, thereby providing the illusion of instant translation necessary in a truly integrated programming environment.

- Support for *interactive execution*, in order to exploit the special nature of an IPE to speed the debugging process.

The performance of language implementations generated from denotational semantics is notoriously poor. Denotational definitions of programming languages use higher-order functions extensively, requiring the use of a specification language that supports the definition, creation, and manipulation of such functions; usually, the specification language chosen is some variant of the λ-calculus. In many compiler-generator systems, programs are translated into a λ-calculus expression, which is then reduced to normal form; this reduction process is extremely slow, and is the source of much of the inefficiency in compiler generators based on denotational semantics.

In our system, called EDS, we have sacrificed some generality for efficiency. Noting that denotational definitions of typical programming languages manipulate functions in limited ways, we extended a general-purpose programming language (Modula-2) to support these limited function manipulations. We were thus able to use this extended language as the specification language for the semantic functions in a denotational definition, avoiding the overhead associated with the overly general function manipulations permitted in the λ-calculus.

EDS-generated interpreters translate programs into denotations that are represented as linked structures containing pointers to the compiled code of denotation functions. This representation affords several advantages:

- The use of pointers to shared copies of the code for denotation functions reduces the space required for the executable representation of a program to a few pointers per construct — comparable to that of a parse tree.

- The denotation functions are compiled once, at system generation time; EDS-generated interpreters can thus translate programs rapidly while maintaining reasonable execution efficiency.

- The linked structure, which corresponds to the flow graph of the program, is easy to maintain incrementally during program modification, thus supporting the instant execution paradigm we desire.

Program execution in EDS is accomplished by executing the functions pointed to by the nodes of the flow graph. These functions invoke appropriate continuations when they have completed their computation; the flow of control is determined by the structure of the flow graph, together with conditional branching functions; no separate interpreter is required. Continuation invocation, the last action performed by a denotation function, provides a natural place for the run-time system (and thereby the user) to interact with a running program. The close correspondence between the flow graph of a program and its syntax tree can be exploited to provide syntax-directed debugging functions, such as flow tracing and breakpointing. If provision is made for the selection of particular continuation links, additional features such as real-time monitoring of variable updates can be provided.

5.2. The Implementation

In order to demonstrate the feasibility of our approach, we have constructed prototype implementations of the major components of EDS. Initial experiments with hand-coded denotation functions verified the advantages of the proposed executable representation for direct semantics, and served to clarify many of the issues involved in simulating function manipulations in a language that did not support functions as first-class objects.

These experiments led to a type-checker and translator for domain definitions and domain element manipulations, which, combined with scanner and parser generators, formed a simple generator for stand-alone interpreters using direct semantics. This stand-alone generator was later extended to support specifications using continuation

semantics, and combined with the Synthesizer Generator [Reps84] to produce a complete IPE generator.

The type-checker and translator together comprise about 4000 lines of Modula-2 code, including interfaces to the scanner and parser generators. We were able to use existing parser and scanner generators, together with generic drivers for the resulting tables, resulting in significant savings in code and effort. More savings result from the use of a "high-level" output language, Modula-2, allowing much processing to be off-loaded to the Modula-2 compiler.

The type-checker and translator process EDS input specifications at about 600 lines per minute on a VAX 11/780, discounting the time required to compile the resulting Modula-2 code. The complete specification for L is translated in 16.2 seconds, with the resulting Modula-2 code requiring 32.5 seconds to compile. The above figures do not take into account time for compiling predefined libraries and interfaces to the other parts of the system; this time is independent of specification size, and most of the auxiliary modules may be pre-compiled, independently of the specification.

Since EDS and the Synthesizer Generator were developed independently, the amalgam was not as complete as might be achieved with a complete IPE generator system developed from the beginning to incorporate the features of both systems. In particular, the internal representations of primitive domains such as integers and identifiers differed significantly between the two systems, requiring some conversion at runtime. In addition, we chose to define a flow graph constructor in SSL, the Synthesizer Generator specification language, using a hand-written attribute grammar to construct the graph. In principle, such a flow-graph constructor should be derivable from a Plumb-like specification of the flow-graph components of the various constructs in the language, but the nature of the output specification (in this case, the attribute

grammar) is specific to the particular LBE generator being used. Section 5.4 on future work describes our vision of a completely integrated generator that would incorporate such a translator.

Despite the incompatibilities between EDS and the Synthesizer Generator, we were able to generate a complete IPE for a simple language (the language L of chapter 2, with **if** and **while** statements) and incorporate flow tracing and breakpointing features into the generated environment. The resultant environment suffered no subjective editing response degradation (as compared to a syntax-only editor), and provided reasonable execution speed — with a static environment and flow tracing disabled, programs executed at about 10K denotations per second, or about one third as fast as the Berkeley Pascal interpreter **px** interprets equivalent Pascal programs. This implementation did not optimize monitor overhead, invoking the monitor on every continuation initiation, and using subroutine calls with stack arguments rather than tail-recursive jumps with global arguments, and consequently monitor and function call overhead accounted for over 60 percent of the execution time; incorporating the optimizations described in section 4.3.2 should reduce this overhead substantially and make the generated environment comparable to **px**. Of course, L is far from being Pascal. How would the performance of an EDS-generated environment change in the transition to a full-scale language?

The execution speed attained using the flow graph executable representation, which we measured at 10K denotations per second, is relatively independent of the actual contents of the denotation, being dominated by the overhead of setting up the environment and arguments for a denotation, and then actually calling the denotation; in comparison, the time taken to execute (say) an addition is negligible. More complex languages, such as Pascal, will require more complex denotations, as well as more

denotations per statement; as an example, a procedure call would, depending on how we wrote our definition, require either one denotation that performed all the necessary argument and local variable stack manipulations as well as the saving of the return continuation, or a sequence of simpler denotations. In either case, the time taken to perform the procedure call would be more than that taken to perform an addition, by about the same factor as would be encountered in compiled code. We therefore expect the performance of an EDS-generated interpreter to remain comparable to that of **px** when both process programs of similar complexity, at least as regards control flow.

From the point of view of EDS, a more significant difference between L and Pascal is the complexity of types and environments encountered in Pascal. Although environments implemented using shallow binding, as used in interpreters for languages with dynamic scoping, and also in Mughal's system [Mughal85] reduce the overhead of variable access in many cases, they do little to reduce the cost of a record field access. In compiled code, record field accesses are essentially free at run-time, given suitable addressing modes, but in a denotational semantics such a field access will involve determining the base location of the record via an environment lookup, determining the field offset via another environment lookup (relative to the result of the first lookup, since field names are not necessarily unique across records), and then a calculation of the resultant location; each of these steps will also involve the overhead of invoking a denotation.

In the case of a language like Pascal, the second environment lookup above is unnecessary — record field offsets are static properties of the program, and could be placed as attributes of the node in question prior to execution. Unfortunately, properties such as this are not easily deduced from the denotational definition of the language. However, for a generated environment to achieve execution efficiency com-

parable to that of a hand-coded interpreter, this and other compile-time simplifications must be done at translation time, rather than deferred until run-time; the same holds true for type-checking, in languages where type is a static property.

In section 5.4 on future work, we present some ideas on how a hybrid system, using an attribute grammar front end coupled with an interpreter generated from denotational semantics might be able to accomplish these simplifications.

5.3. Comparison with related work

There has been other work with goals similar to ours; in order to place our work in perspective, we will briefly summarize some of this other work and compare it with ours.

5.3.1. Other Integrated Programming Environments

Throughout the thesis, we have been using the Cornell Program Synthesizer as an example of the type of integrated programming environment that we wish to generate. There are other notable integrated programming environments, and we describe some of these below. Although the systems described are *environments* rather than *generators*, they are interesting as comparison points for automatically generated environments.

5.3.1.1. Magpie

Magpie [Delisle84, Schwartz84] is an integrated programming environment for Pascal based on incremental compilation. This incremental compilation allows the programmer to specify debugging actions in Pascal, eliminating the need for a separate debugging language.

Magpie's user interface is designed to remove the separation between program development tools; all commands that make sense in a particular context are made available within that context, regardless of whether they are associated with editing or debugging.

Magpie uses display windows (called browsers) to provide access to both the program and its execution state. Debugging actions are specified by entering Pascal statements that are executed immediately in the context of the suspended execution state. In addition, the user may specify *demons*, written in Pascal, that monitor run-time events such as assignment to a variable or invocation of a procedure.

Since Magpie runs on a powerful single-user workstation, it is able to utilize programmer "think time" to compile programs in the background. Translation is performed in units of procedures, and with incremental linking via a stub linked at the entry point of any procedure for which the machine code has not yet been obtained. Execution starts immediately in response to user command, and if execution reaches an un-translated procedure, the stub invokes the translator to produce machine code for it; if this translation is unsuccessful, a run-time error occurs and execution is suspended.

Magpie supports execution and debugging functions by instrumenting translated code with debugging code, which may invoke a demon or run-time support routine. In order to reduce recompilation, which may invalidate execution state, Magpie inserts *debugging hooks* in functions, which allow debugging to be turned on and off without regenerating code.

5.3.1.2. DICE

The DICE (Distributed Incremental Compiling Environment) system [Fritzson84] is an integrated programming environment, based on incremental compilation, that

runs on a host computer and supports the development of programs that run on a target computer connected to the host.

The DICE system performs statement level incremental compilation on Pascal, and permits incremental recompilation even after changes to global declarations. The system is based solely on compilation technology, with no interpretation, even for debugging. Databases of cross-reference and static analysis information are an integral part of the DICE system, and are used to support incremental recompilation as well as some debugging facilities. The cross-reference database is used to identify statements that need recompilation after changes to global declarations.

The DICE system uses the debugger as the focus of integration; all system capabilities are available from the debugger, such as editing and interrogation of the static analysis database. The debugger works in units of statements, so that breakpoints may be placed at statement boundaries only, and the target machine state is also defined only at these boundaries.

The parser, pretty-printer and editor in the DICE system are table-driven, while the code generator is based on that of the portable C compiler [Johnson79]; static semantics are hand programmed, as is the debugger.

5.3.2. Action Equations

Kaiser [Kaiser85, Kaiser86] proposes the use of *action equations*, which are claimed to extend attribute grammars to specify the run-time semantics of languages, by supporting the expression of *history* or dynamic properties. Instead of attribute definitions, Kaiser embeds *equations* in an event-driven architecture. Events activate equations, and active equations are evaluated according to dependencies derived from the equations. Some equations are not associated with any event, and are thus in effect

at all times — these equations correspond to conventional attribute grammar definitions.

The advantages claimed for the action equation paradigm are support for multiple events (as opposed to a single *change* event in attribute grammars), and support for non-applicative mechanisms. Correspondences are drawn between *events* and messages in an object-oriented language, and between the equations activated by an event and the method invoked by a message. In order to simulate message passing, equations may *propagate* events as well as apply functions.

Action equations are implemented by translating them into a combination of dependency graphs and procedures written in a special tree-oriented programming language. The dependency graphs are used to determine the order in which to invoke the procedures corresponding to active equations.

The goals of Kaiser's work are almost identical to ours; the primary difference is the vehicle chosen to achieve these goals. We have adopted a proven technique, denotational semantics, and adapted it to the incremental and interactive nature of IPEs; Kaiser has proposed a new specification technique. Although [Kaiser86] mentions that action equations have only been used to specify toy environments, it claims that they can be used to implement run-time facilities for programming environments supporting realistic languages. Denotational definitions of realistic programming languages such as Pascal already exist [Tennent78].

5.3.3. The PSG programming system generator

The PSG programming system generator [Bahlke85] developed at the Technical University of Darmstadt combines the use of context conditions [Snelting86] for static semantic specification with a denotational description for dynamic semantics to pro-

duce a complete language description. This description is processed by PSG to produce a complete programming environment.

The denotational part of the language definition is written in a functional language based on type-free λ-calculus. This language supports basic data types including integer, real, boolean, string and identifier, and the structured types list/tuple and map, as well as higher-order functionals of arbitrary rank. Arguments to function applications are evaluated *call-by-need*, which provides advantages of both call-by-value and call-by-name, and is a correct implementation of recursion allowing non-strict functions. Call-by-need also allows visible side-effects within the environment, which are used to support interactive output.

The translator in a PSG environment can perform incremental compilation, but only in cases where the abstract syntax tree is created top-down by a series of refinement steps; in other words, incremental compilation is possible, but not incremental *re*-compilation. This incremental compilation capability also permits execution of incompletely specified programs, with execution being suspended when an unexpanded construct is encountered, and resumed when the required fragment has been supplied.

Execution in PSG systems is by λ-calculus reduction, using closures and environments as in the SECD machine. During execution, user interaction for input, output, expanding unspecified constructs and supplying values for uninitialized variables is permitted.

EDS was developed independently of PSG. PSG is a complete environment generator, but suffers from the performance problems associated with denotational semantics based compilers using direct λ-calculus reduction. The fragment concept, used throughout PSG, supports run-time expansion of incomplete programs, and user-

supplied values for uninitialized variables. Other debugging functions provided in EDS, such as flow tracing and breakpointing, are not supported by PSG.

5.3.4. MELA

The MELA metalanguage, developed at the University of Pisa [Ambriola85], is a functional language that supports the definition of higher order functions and domains required for denotational specifications. MELA descriptions are automatically translated to a conventional executable language through a series of correctness-preserving transformations, resulting in an interpreter for the language described.

This interpreter may be combined with a structure editor to form a complete programming environment, but such an environment is not truly integrated; no mention is made in [Ambriola85] of incremental processing, nor is interactive execution specifically addressed.

5.3.5. Reppy and Kintala

Reppy and Kintala [Reppy84] proposed the use of denotational semantics to specify the run-time semantics of languages, and thereby generate execution facilities for IPEs. They also noted the correspondence between Sethi's flow graphs [Sethi83] and *code trees* in the Cornell Program Synthesizer [Teitelbaum81] and suggested that code trees could be generated from denotational specifications by techniques similar to Sethi's.

Much of the inspiration for our work has come from that of [Reppy84] and there are many similarities between the two. One major difference is in the choice of executable representation: where [Reppy84] uses code trees containing op-codes for an abstract machine, together with an interpreter for the corresponding virtual machine, we use pointers to compiled denotation functions, thus dispensing with the need for an

interpreter (although an execution monitor is still needed to provide debugging support).

Although Reppy and Kintala suggest the incorporation of interactive execution facilities into the generated environment, the presentation in [Reppy84] does not develop this subject fully. We believe that our model of execution monitoring, where debugging features are incorporated as store-transforming functions interposed before continuation invocations, gives a conceptually clean framework for the specification of these features, while permitting their efficient implementation.

5.3.6. Mughal

Like [Reppy84], the system described by Mughal [Mughal85] also uses a control-flow graph as the executable representation of a program, but once again the contents of the flow graph differentiate it from EDS. In [Mughal85], the instruction set recognized by the interpreter is defined through the use of SSL, the specification language for the Synthesizer Generator [Reps84]. The domain of op-codes is thus defined as a *phylum* (in Synthesizer Generator terminology) with an *operator* corresponding to each op-code. This representation provides a consistent abstract machine within which to specify the run-time semantics of a language. The abstract machine can be extended by adding new operators to the CODE phylum, provided the interpreter specification is augmented to recognize and implement the new operators.

The use of an interpreter for the abstract machine code used as the executable representation of programs results in much slower execution than EDS, which uses pointers to compiled denotation functions.

Although [Mughal85] uses a high-level specification language (SSL) to define the interpreter for the abstract machine code, the code associated with a specific construct

is still specified directly rather than being generated from a high-level definition of the run-time semantics of the construct. Assuming a suitable choice of abstract machine operations, however, it seems that Sethi's flow graph construction technique can be adapted for use here too.

As with [Reppy84] and EDS, the use of an attribute grammar to define the translation from syntax tree to executable representation yields incremental translation automatically, given the incremental attribute updating algorithms available, such as [Reps83] and [Johnson83]. Interactive execution and debugging are also discussed in [Mughal85], where the code block is taken as the unit of execution, analogous to the continuation invocation in EDS.

5.4. Future work

Through the design and prototype implementation of EDS, we have shown the feasibility of using denotational semantic specifications to generate execution facilities for integrated programming environments. We believe this is a promising approach to producing complete IPEs for a variety of languages, and should be investigated further.

A key issue in the design of an integrated generator will be the division of functionality between static and dynamic semantics. As we alluded to earlier, such static properties of programs as type-checking and record field offset determination should be done at translation time in order to achieve the best performance possible. In addition, static type checking helps find errors earlier in the program development process, which is one of the strongest arguments in favor of static typing systems.

It seems then, that efficiency considerations as well as error detection are best served by separating static and dynamic aspects of a language in an implementation. On the other hand, an attraction of denotational semantics is its ability to encompass all

aspects of a language semantics within a single framework. In addition, there are denotational semantics for several programming languages, and we would like to use these specifications with as little modification as possible. How then do we resolve these two competing goals?

Let us first describe our vision of what a hybrid environment for a language like Pascal would look like, and then outline some approaches that could lead to reconciliation of such an environment with the desire to use standard denotational semantics.

The front-end of a hybrid Pascal environment would perform all the semantic processing already performed by language-based editors, such as type checking and detection of undeclared variables. In addition, we would use the front-end specification mechanism (say attribute grammars) to implement two translation functions — the creation of the program denotation, and the calculation of the static components of the environment.

Thus we would expect the attribute rules to supply each identifier usage with an attribute corresponding to its associated environment value, assuming that that value is statically determined. An example of such a statically determined value would be record field offsets. Activation record offsets for variables could also be assigned statically.

Although these values can be assigned statically, a naive implementation using an attribute grammar might be extremely inefficient. For example, consider allocating activation record offsets depending on the lexical order of the variable declarations in the procedure. In that case, addition of a new variable declaration preceding all the others would cause re-assignment of the offsets of all the variables in the procedure, which would propagate to all uses of variables in the procedure. Such propagation is clearly undesirable.

We can avoid this propagation by allocating activation record offsets on the basis of the *temporal* order in which variable declarations are introduced (treating modification as deletion followed by insertion). With this scheme, the activation record offset of a variable is never changed except as a result of changes to the variable declaration (or other declarations it depends on explicitly). This scheme has also introduced some order-dependence into our system, which we had earlier deemed undesirable; however, the order-dependence is very closely contained, by concealing it in a single attribute evaluation function, which determines the offset attribute of a variable as a function of the size of the variable. This attribute evaluation function itself violates the functional nature of attribute grammars, since its return value may differ on different calls, even though called with the same argument. This violation is benign, however, since incremental re-evaluation will not invoke the function, assuming its result will be unchanged, since its argument has not changed, which is exactly what we want; in the case where the attribute re-evaluation mechanism mistakenly invokes the function even though its argument value has not changed, we still maintain correctness, although some spurious attribute re-evaluation will be triggered as a result of the change in the value of the function result.

Given that the front-end of our environment has performed all this checking and processing, the denotational semantics from which the back-end is generated can be simplified and made more efficient. For example, type checking need no longer be done at run time, and environment access is simplified to be the retrieval of an attribute value from the identifier to which the environment is to be applied. For procedure calls, if local variables are allotted activation record offsets, then the environment manipulations necessary at procedure entry and exit are simply the stack manipulations necessary in conventional compiled code; the size of a procedure's activation record

will be an attribute of the procedure header.

We now face the problem alluded to earlier — how do we derive these separate specifications from a denotational semantics (and possibly an attribute grammar specification) of the language, and how do we ensure that the composite does indeed specify a correct implementation of the language?

Although we do not at present have an answer to this question, we can outline some possible approaches:

- The most ambitious, and probably least feasible, scheme is to deduce automatically both parts of the specification from a denotational semantics for the language. A general method of separating the two aspects seems unlikely — implementation tricks such as static chains are extremely hard to deduce from the denotational semantics of a language.
- At the other end of the spectrum is verification; given a denotational semantics, attribute grammar, and modified denotational semantics, the system verifies that the attribute grammar and the modified denotational semantics together are equivalent to the standard denotational semantics.
- A variety of intermediate approaches is possible. Given a standard denotational semantics and an attribute grammar, the system can deduce a modified denotational semantics. The variety of approaches arises from the degree of assistance that the language specifier is required to provide during the transformation process.

In a more mundane vein, there are other optimizations that could be performed to improve the performance of EDS-generated systems. The need to call a continuation for every construct in the language, no matter how trivial, is a major source of over-

head. Sethi [Sethi83] uses a graph reducer to coalesce linear sequences of instructions into basic blocks; a similar technique could be used in EDS, but would obscure the correspondence between syntax tree and executable representation, complicating incremental translation and debugging.

In conclusion, then, we have presented a system that can generate interpreters based on denotational semantics. Although the generated interpreters have performance comparable to that of hand-coded interpreters for control flow aspects of simple languages, work needs to be done to improve the performance of the interpreters with regard to optimization of static properties. We feel that a hybrid approach, combining the best features of attribute grammars with the techniques developed in EDS, holds the key to generating interpreters for realistic programming languages that have performance comparable to hand-coded systems.

References

[Ambriola84] Ambriola, Vincenzo, Gail E. Kaiser, and Robert J. Ellison, "An Action Routine Model for ALOE," Technical Report CMU-CS-84-156, CMU Department of Computer Science, August 1984.

[Ambriola85] Ambriola, Vincenzo and Carlo Montangero, "Automatic Generation of Execution Tools in a GANDALF Environment," *The Journal of Systems and Software* 5:2 (May 1985), pp. 155-171.

[Appel85] Appel, Andrew W., "Semantics-Directed Code Generation," *Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages*, January 25-27, 1985, pp. 315-324.

[Bahlke85] Bahlke, Rolf and Gregor Snelting, "The PSG - Programming System Generator," *Proc. ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, June 25-28, 1985, pp. 28-33.

[Delisle84] Delisle, Norman M., David E. Menicosy, and Mayer D. Schwartz, "Viewing a Programming Environment as a Single Tool," *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 23-25, 1984, pp. 49-64.

[Demers82] Demers, Alan, *Course notes for CS 611 - Advanced Programming Languages*, Cornell University, 1982.

[Dewan86] Dewan, Prasun, "Automatic Generation of User Interfaces," Ph.D. Thesis, University of Wisconsin - Madison, September 1986.

[Fischer84] Fischer, Charles N., Gregory F. Johnson, Jon Mauney, Anil Pal, and Daniel L. Stock, "The Poe Language-Based Editor Project," *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 23-25, 1984, pp. 21-29.

[Fritzson84] Fritzson, Peter, "Preliminary Experience from the DICE System, A Distributed Incremental Compiling Environment," *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 23-25, 1984, pp. 113-123.

[Gordon79] Gordon, M. J. C., *The Denotational Description of Programming Languages*, Springer-Verlag, New York, 1979.

[Habermann82] Habermann, A. N. and D. Notkin, "The Gandalf Software Development Environment," in *The Second Compendium of Gandalf Documentation*, Carnegie-Mellon University, Computer Science Department, May 1982.

[Henson82] Henson, Martin C. and Raymond Turner, "Completion Semantics and Interpreter Generation," *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, January 25-27, 1982, pp. 242-254.

[Horwitz85] Horwitz, Susan and Tim Teitelbaum, "Relations and Attributes: A Symbiotic Basis for Editing Environments," *Proceedings of the ACM SIGPLAN 85 Symposium on Language Issues in Programming Environments*, June 25-28, 1985, pp. 93-106.

[Jensen78] Jensen, K. and N. Wirth, *Pascal User Manual and Report, Second Edition*, Springer-Verlag, New York, 1978.

[Johnson83] Johnson, Gregory F., "An Approach to Incremental Semantics," Ph.D. Thesis, University of Wisconsin - Madison, August 1983.

[Johnson79] Johnson, S. C., "A tour through the portable C compiler," in *Unix Programmers manual, 7th edition*, January 1979.

[Kaiser85] Kaiser, Gail E., "Semantics for Structure Editing Environments," Ph. D. Thesis, Carnegie-Mellon University, May 1985.

[Kaiser86] Kaiser, Gail E., "Generation of Run-Time Environments," *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, June 25-27, 1986, pp. 51-57.

[Kernighan78] Kernighan, Brian W. and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall, Inc., Englewood Cliffs, New Jersey,, 1978.

[Knuth68] Knuth, Donald E., "Semantics of Context-Free Languages," *Mathematical Systems Theory* 2:2 (June 1968), pp. 127-145.

[Linton84] Linton, Mark A., "Implementing Relational Views of Programs," *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 23-25, 1984, pp. 132-140.

[Medina-Mora82] Medina-Mora, Raul, "Syntax-Directed Editing: Towards Integrated Programming Environments," PhD Thesis, Carnegie-Mellon University, March 1982.

[Mosses76] Mosses, P. D. , "Compiler Generation Using Denotational Semantics," pp. 436-441 in *Mathematical Foundations of Computer Science, Lecture Notes in Computer Science*, Springer-Verlag, New York, 1976.

[Mughal85] Mughal, K. A., "Control Flow Aspects of Generating Runtime Facilities for Language-Based Programming Environments," *Proceedings of the 1985 IEEE Computer Society Conference on Software Tools*, April 15-17, 1985.

[Naur63] Naur, P., "Revised Report on the Algorithmic Language ALGOL 60," *Communications of the ACM* 6:1 (January 1963), pp. 1-17.

[Paulson82] Paulson, Lawrence, "A Semantics-Directed Compiler Generator," *Conference Record of the 9th Annual ACM Symposium on Principles of Programming Languages*, January 25-27, 1982, pp. 224-233.

[Reppy84] Reppy, J. H. and C. M. R. Kintala, "Generating Execution Facilities for Integrated Programming Environments," Technical Memorandum, A.T.&T. Bell Laboratories, Murray Hill, N.J., March 30, 1984.

[Reps83] Reps, Thomas, Tim Teitelbaum, and Alan Demers, "Incremental context-dependent analysis for language-based editors.," *ACM TOPLAS* 5:3 (July 1983), pp. 440-477.

[Reps84] Reps, Thomas and Tim Teitelbaum, "The Synthesizer Generator," *Proc. ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, April 23-25, 1984, pp. 42-48.

[Reps86] Reps, Thomas, Carla Marceau, and Tim Teitelbaum, "Remote Attribute Updating for Language-Based Editors," *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, January 13-15, 1986, pp. 1-13.

[Schmidt85] Schmidt, David A., "Detecting Global Variables in Denotational Specifications," *ACM TOPLAS* 7:2 (April 1985), pp. 299-310.

[Schwartz84] Schwartz, Mayer D., Norman M. Delisle, and Vimal S. Begwani, "Incremental Compilation in Magpie," *Proceedings of the SIGPLAN '84 Symposium on Compiler Construction*, June 17-22, 1984, pp. 122-131.

[Scott71] Scott, D.S. and C. Strachey, "Towards a mathematical semantics for computer languages," *Proceedings of the Symposium on Computers and Automata*, Polytechnic Institute of Brooklyn Press, April 1971, pp. 19-46.

[Sethi83] Sethi, R., "Control Flow Aspects of Semantics Directed Compiling," *ACM TOPLAS* 5:4 (October 1983), pp. 554-595.

[Snelting86] Snelting, Gregor and Wolfgang Henhapl, "Unification in Many-Sorted Algebras as a Device for Incremental Semantic Analysis," *Conference Record of the 13th Annual ACM Symposium on Principles of Programming Languages*, January 13-15, 1986, pp. 229-235.

[Stoy77] Stoy, Joseph E., *Denotational Semantics: The Scott-Strachey Approach to Programming Language Theory*, The MIT Press, Cambridge, Massachusetts, and London, England, 1977.

[Strachey74] Strachey, C. and C.P. Wadsworth, "Continuations - a Mathematical Semantics for Handling Full Jumps," Technical Monograph PRG-11, Programming Research Group, University of Oxford, 1974.

[Teitelbaum81] Teitelbaum, T. and T. Reps, "The Cornell Program Synthesizer: A Syntax-Directed Programming Environment," *Communications of the ACM* 24:9 (September, 1981), pp. 563-573.

[Tennent78] Tennent, R. D., "A Denotational Definition of the Programming Language Pascal," Technical Report, Programming Research Group, University of Oxford, April 1978.

[Tennent81] Tennent, R. D., *Principles of Programming Languages*, Prentice-Hall International, Inc., London, 1981.

[Tichy86] Tichy, Walter F., "Smart Recompilation," *ACM TOPLAS* 8:3 (July 1986), pp. 273-291.

[Wand82] Wand, Mitchell, "Deriving Target Code as a Representation of Continuation Semantics.," *ACM TOPLAS* 4:3 (July 1982), pp. 496-517.

[Wirth83] Wirth, Niklaus, *Programming in Modula-2, second edition*, Springer-Verlag, Berlin Heidelberg New York, 1983.