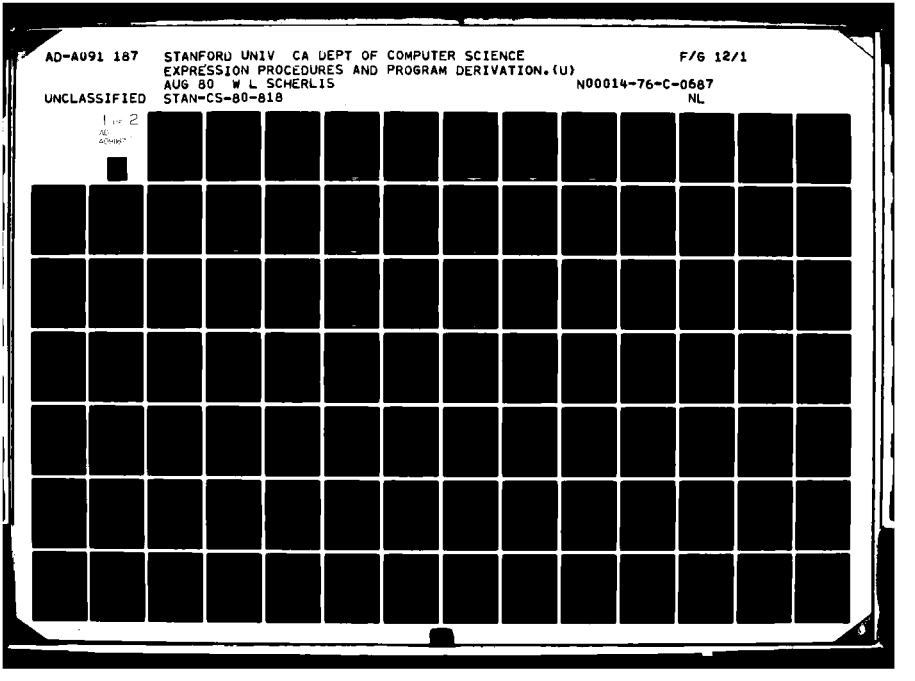

AD-A091 187 STANFORD UNIV CA DEPT OF COMPUTER SCIENCE F/G 12/1
EXPRESSION PROCEDURES AND PROGRAM DERIVATION.(U)
AUG 80 W L SCHERLIS
UNCLASSIFIED STAN-CS-80-818 N00014-76-C-0687
NL
1 OF 2

Stanford Artificial Intelligence Laboratory
Memo AIM-341

August 1980

Department of Computer Science
Report No. STAN-CS-80-818

12

LEVEL II

EXPRESSION PROCEDURES AND PROGRAM DERIVATION

by

William L. Scherlis

Research sponsored by

National Science Foundation
Office of Naval Research
United States Air Force

COMPUTER SCIENCE DEPARTMENT
Stanford University

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE	READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER: STAN-CS-80-818, (AIM-341) — 2. GOVT ACCESSION NO.: AD-A091 187	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle): Expression Procedures and Program Derivation	5. TYPE OF REPORT & PERIOD COVERED: technical, August 1980
	6. PERFORMING ORG. REPORT NUMBER: STAN-CS-80-818 (AIM-341)
7. AUTHOR(s): William Louis Scherlis	8. CONTRACT OR GRANT NUMBER(s): N0001 4-76-C-0687
9. PERFORMING ORGANIZATION NAME AND ADDRESS: Department of Computer Science, Stanford University, Stanford, California 94305 USA	10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS: 99
11. CONTROLLING OFFICE NAME AND ADDRESS: Office of Naval Research, Department of the Navy, Arlington, Virginia 22217	12. REPORT DATE: Aug[redacted]80 — 13. NO. OF PAGES: 178
14. MONITORING AGENCY NAME & ADDRESS (if diff. from Controlling Office): ONR Representative - Philip Surra, Durand Aeromautics Building, Room 165, Stanford University	15. SECURITY CLASS. (of this report): Unclassified
	15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this report)

Approved for public release; distribution unlimited.

9 - Ph.D. thesis

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

(see other side)

DD FORM 1 JAN 73 1473

EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED 094120

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

19. KEY WORDS (Continued)

20 ABSTRACT (Continued)

We explore techniques for systematically deriving programs from specifications. The goal of this exploration is a better understanding of the development of algorithms. Thus, the intention is not to develop a theory of programs, concerned with the analysis of existing programs, but instead to develop a theory of programming, dealing with the process of constructing new programs. Such a theory would have practical benefits both for programming methodology and for automatic program synthesis.

We investigate the derivation of programs by program transformation techniques. By expanding an ordinary language of recursion equations to include a generalized procedure construct (the *expression procedure*), our ability to manipulate programs in that language is greatly facilitated. The expression procedure provides a means of expressing information not just about the properties of individual program elements, but also about the way they relate to each other.

A set of three operations --- abstraction, application, and composition ---for transforming programs in this extended language is presented. We prove using operational semantics that these operations preserve the strong equivalence of programs. The well-known systems of Burstall and Darlington and of Manna and Waldinger are both based on an underlying rule system that does not have this property.

Our transformation system is illustrated with several small examples, which are examined in detail to give insight to the heuristic problems of program development. A tactic of *program specialization* is shown to underlie many of these derivations. While we present no implemented system, some consideration is given to issues related to the development of programming tools.

The practical value of our approach is demonstrated in the systematic development of a partial family tree of parsing algorithms, similar in spirit to the treatment of sorting by Darlington and by Green and Barstow. Several intricate parsing algorithms (Earley's, Cocke-Younger-Kasami, and Top-Down) are derived in detail.

Stanford Artificial Intelligence Laboratory
Memo AIM-341

August 1980

Department of Computer Science
Report No. STAN-CS-80-818

EXPRESSION PROCEDURES AND PROGRAM DERIVATION

by

William L. Scherlis

ABSTRACT

We explore techniques for systematically deriving programs from specifications. The goal of this exploration is a better understanding of the development of algorithms. Thus, the intention is not to develop a theory of programs, concerned with the analysis of existing programs, but instead to develop a theory of programming, dealing with the process of constructing new programs. Such a theory would have practical benefits both for programming methodology and for automatic program synthesis.

We investigate the derivation of programs by program transformation techniques. By expanding an ordinary language of recursion equations to include a generalized procedure construct (the *expression procedure*), our ability to manipulate programs in that language is greatly facilitated. The expression procedure provides a means of expressing information not just about the properties of individual program elements, but also about the way they relate to each other.

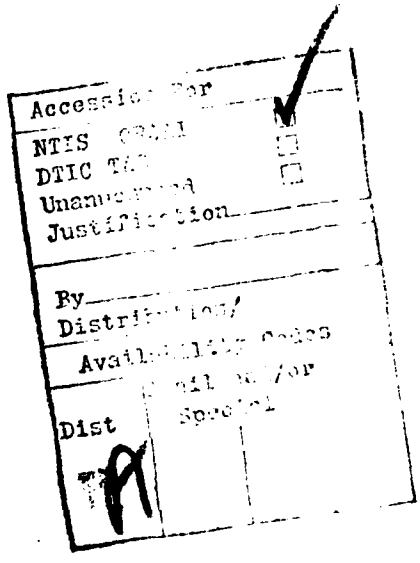
A set of three operations --- abstraction, application, and composition ---for transforming programs in this extended language is presented. We prove using operational semantics that these operations preserve the strong equivalence of programs. The well-known systems of Burstall and Darlington and of Manna and Waldinger are both based on an underlying rule system that does not have this property.

Our transformation system is illustrated with several small examples, which are examined in detail to give insight to the heuristic problems of program development. A tactic of *program specialization* is shown to underlie many of these derivations. While we present no implemented system, some consideration is given to issues related to the development of programming tools.

The practical value of our approach is demonstrated in the systematic development of a partial family tree of parsing algorithms, similar in spirit to the treatment of sorting by Darlington and by Green and Barstow. Several intricate parsing algorithms (Earley's, Cocke-Younger-Kasami, and Top-Down) are derived in detail.

This thesis was submitted to the Department of Computer Science and the Committee on Graduate Studies of Stanford University in partial fulfillment of the requirements for the degree of Doctor of Philosophy.

This research was supported by the National Science Foundation, Department of the Navy Contract N00014-76-C-0687, and United States Air Force. The views and conclusions contained in this document are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of Stanford University, or any agency of the U. S. Government.



by

William L. Scherlis

EXPRESSION PROCEDURES AND PROGRAM DERIVATION

A DISSERTATION

SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE

AND THE COMMITTEE ON GRADUATE STUDIES

OF STANFORD UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

DOCTOR OF PHILSOPHY

By

William Louis Scherlis

August 1980

Acknowledgements

I am deeply grateful to:

Zohar Manna, my advisor, for his insight, enthusiasm, and patience. Without his encouragement and support I could not have brought this work to completion.

Richard Waldinger, for his careful reading and insightful comments.

Don Knuth, who contributed to both the form and content of this thesis. I greatly enjoyed using his TEX typesetting system.

Richard Weyhrauch, for good advice and many interesting conversations.

Tom Cheatham, for his encouragement and understanding over the years.

Rod Burstall and John Darlington, who inspired me to work on this topic.

Pierre Wolper, who helped me debug the dire technical chapter.

My friends Martin Brooks, Denny Brown, Dick Gabriel, Chris Goad, Greg Nelson, Jorge Phillips, Marty Raim, Richard Schwartz, Dan Sleator, who have all helped me in various ways in the course of this work.

Linnea.

And my family, to whom I dedicate this thesis.

This research was supported in part by an IBM Predoctoral Fellowship, in part by the National Science Foundation under grant MCS-79-09495, in part by the Office of Naval Research under contract N00014-76-C-0687, and in part by the Air Force Office of Scientific Research under contract F30602-78-0099.

Table of Contents

Chapter 1

Program Manipulation

Many modern programming language designs are based on the principle that programmers should minimize the amount of interconnection between the parts of their programs. This principle of modularity is strongly reflected in the design of the various packaging or modularization features in these languages. (See, e.g., [Jensen74], [Dahl72], [Ichbiah79], [Wirth76], [Wulf76], and [Liskov77].) Programs that are structured according to these principles are generally recognized to be easier to understand, debug, and maintain than equivalent programs written in a "nonstructured" way.

Unfortunately, such structuring of programs is often at the expense of their ultimate efficiency. Many optimizations that occur in the algorithm design or programming process are the result of the recognition of potential computational sharing across package boundaries. In order to take advantage of such sharing, the "structured" program must be modified to allow more information to cross package boundaries, contradicting the fundamental design principle of modularity.

Thus unless he writes unusually elegant programs, a programmer is faced with a choice between efficiency and lucidity: If he desires a very efficient program, he may find that the interaction between various packages must be so great that it would be best to do away with the package boundaries altogether. If he desires a nicely structured program on the other hand, he may find that the cost of maintaining structure, in redundant computation or storage space, to be excessive.

We propose that in a framework of program derivation, it is possible to have the best of both worlds, deriving an efficient highly connected program from a less efficient, but more lucid, initial program.

In this thesis we explore methods for the systematic derivation of programs. The goal of this exploration is a better understanding of the problems of program development, leading to an improved methodology and to improved tools. Our approach is based on a system of strong-equivalence preserving program transformation rules, and a technique for using them called *internal specialization*.

The specialization tactic follows naturally from the observation that program development is a directional process, in which the degree of interconnection and interdependence between the parts of a program is increased to obtain improvements in efficiency.

1. Internal Specialization

It is often most natural to approach a new computational problem by viewing it as a special case of a previously solved, more general problem. This gives the programmer or algorithm designer the advantage of the knowledge that at least there is some solution to his problem, though it may not be as efficient as he would like. The process of programming or algorithm design then becomes one of *specialization*, in which a general algorithm is tailored into a closely fitting solution for the problem at hand.

We take a very broad interpretation of the notion of specialization: One program can be a specialized version of another even though they both compute the same function on the same domain.

A clear conceptual structure in a program implies that there are a minimum of interconnections between the parts of that program. This makes the program easy to understand, but it also implies that the parts of the program must be specified with a minimum amount of contextual reference. By incorporating additional context into the subprograms, however, they can be tuned to the particular subset of tasks that will actually be required. We can view this as an *internal* specialization process, in which generally specified subprograms are tailored into close fitting specific solutions, trading generality and clarity for efficiency.

It is frequently the case that there are several distinct ways in which a program may be internally specialized, resulting in different implementations

of the same initial specification. For example, given a general specification of the sorting problem, different decisions made during the development process lead to different sorting algorithms. Clark and Darlington and, independently, Green and Barstow have developed some very elegant derivation trees for sorting. Such a tree has the following structure: The root of the tree is labeled with a very simple sorting program, characterized by lucidity rather than efficiency. This initial algorithm has a *descriptive* role; it serves as a starting point simply because we accept it as being an adequate description of the problem, and an initial approximation to an efficient solution. As we move downwards in such a tree, the algorithm becomes more and more specialized. The nodes in the tree thus represent decision points, at which a variety of specialization steps may be made. (See [Green78], [Darlington78a], [Clark80], and [Barstow80].)

For example, at one such decision point, a divide-and-conquer tactic is being applied to an array. One possibility is to divide the array by splitting it into two nearly equal-sized segments, and then to treat each segment independently. Another possibility is to split the array into its first element and the rest of the elements. The latter alternative yields algorithms such as insertion sort and selection sort. The former yields quicksort and merge sort.

Such derivation trees reveal much, both about the structure of the individual algorithms and about the way they relate to each other. This is one of the reasons why such work in program derivation can have a significant impact on programming methodology.

We will explore this specialization approach in greater depth, particularly as applied to the case of the derivation of context-free parsing algorithms. Parsing algorithms are more complicated than sorting algorithms, and so this initial effort goes only a part of the way towards the development of a derivation tree as nicely complete as those for sorting. Derivations are given for the Cocke-Younger-Kasami algorithm, Earley's algorithm, and a top-down algorithm.

There are two sources of heuristic difficulty in the specialization process. The first difficulty is in the selection of the initial algorithm: A wrong choice here may mean that the sequence of specialization steps is longer and more complicated than necessary, or even that an adequately efficient specialized algorithm is not practically obtainable. The second difficulty is in the specializa-

tion process itself: Specialization is a sequential process, in which an algorithm is transformed into a more specific one in a series of steps, each step reflecting some observation about the behavior of the algorithm at that point. It sometimes happens that a particular step of specialization relies on an observation involving some deep property of the algorithm. Such "eureka" steps (in the terminology of Burstall and Darlington, [Burstall77]) may require some inspiration to find.

It is safe to say, however, that specialization is usually more heuristically manageable than the inverse process of *generalization*. In the latter case, a programmer does not have the comfort of a worst-case starting point, and may have to confront a much larger space of possibilities. The specialization process has the advantage that it starts from a worst case solution and continues only until an acceptable solution is reached. Generalization is more a process of discovery: There is no guarantee of finding even a worst-case solution.

We can illustrate the idea of specialization with an example. Consider the following program for reversing lists:

$$rev(x) \leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(x)) \circ cons(hd(x), \Lambda).$$

Here, *hd*, *tl*, and *cons* are the usual functions for selecting and constructing lists; Λ (or nil) denotes the empty list, and the infix operator '$\circ$' is another name for the list concatenation function *append*.

Suppose that after we reverse a given list x, we will require only the first element of the result. We describe this situation by using the notation:

$$\begin{array}{rll} \text{Compute:} & hd(rev(x)) & (\text{where } x \neq \Lambda) \\ \text{Given:} & rev(x) \leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(x)) \circ cons(hd(x), \Lambda). & \end{array}$$

(To avoid erroneous cases, we restrict the initial list to be nonempty.) This program computes the desired value by constructing a reversed version of the initial list and then taking the *hd* of the result.

Using the transformation rules we describe below, we can specialize the auxiliary function *rev* to the case in which only the first element of its result is required, producing a new function *hdrev*. We obtain the new program,

$$\begin{array}{rll} \text{Compute:} & hdrev(x) & (\text{where } x \neq \Lambda) \\ \text{Given:} & hdrev(x) \leftarrow \textbf{if } tl(x) = \Lambda \textbf{ then } hd(x) \textbf{ else } hdrev(tl(\ldots)), & \end{array}$$

which produces results identical to those of the initial specification, but without doing any list construction at all.

Similarly, a transformation of the naïve square root program,

$$\begin{array}{l} s(x) \leftarrow q(0, x) \\ q(i, x) \leftarrow \textbf{if } i^2 \leq x < (i+1)^2 \textbf{ then } i \textbf{ else } q(i+1, x) \end{array}$$

over the nonnegative integers, into the equivalent, but more efficient program,

$$\begin{array}{l} s(x) \leftarrow r(0, 1, 3, x) \\ r(i, m, n, x) \leftarrow \textbf{if } x < m \textbf{ then } i \textbf{ else } r(i+1, m+n, n+2, x), \end{array}$$

could be viewed as a process of specialization of the auxiliary function q into a new and more efficient auxiliary function r. Observe that the initial program requires two multiplications and two additions on each iteration, while the derived program requires three additions only.

Finally, we can improve the recursive list reversing program from the initial example,

$$rev(x) \leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(x)) \circ cons(hd(x), \Lambda),$$

by specializing the *recursive instance* of *rev* to obtain

$$\begin{array}{l} rev(x) \leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } rev2(tl(x), cons(hd(x), \Lambda)) \\ rev2(u, v) \leftarrow \textbf{if } u = \Lambda \textbf{ then } v \textbf{ else } rev2(tl(u), cons(hd(u), v)), \end{array}$$

or more simply

$$\begin{array}{l} rev(x) \leftarrow rev2(x, \Lambda) \\ rev2(u, v) \leftarrow \textbf{if } u = \Lambda \textbf{ then } v \textbf{ else } rev2(tl(u), cons(hd(u), v)). \end{array}$$

The initial program is inefficient because of the use of the *append* function: If *cons* is the only available list constructing operator, then *append* must be implemented in terms of *cons*, and it must call *cons* once for each element of its first argument: Thus, reversing a list will entail a number of calls to *cons* proportional to the square of the length of the input list. The final program, however, is linear — requiring a number of *cons*'s equivalent to the length of the input list. Furthermore, because of its iterative structure it can be implemented very efficiently.

2. Outline of This Work

These examples illustrate the potentially broad range of applicability of a general-purpose specialization technique. Our goal in this investigation will be to develop such a technique, and to demonstrate its use in the derivation of complicated algorithms.

The specialization technique we develop is based on a new approach to the transformation of applicative programs: By expanding an ordinary language of recursion equations to include a generalized procedure construct — the *expression procedure* — our ability to manipulate programs in that language will be greatly enhanced. The expression procedure provides a way of expressing information not just about the properties of individual program elements, but also about the way they relate to each other.

We develop in this and the next chapter a system of program transformation rules based on this approach: In the remainder of this chapter, we give an informal example to illustrate the use of expression procedures in program derivation, and we describe the four basic program transformation rules. The second chapter, which may be omitted on first reading, is devoted to a rigorous statement of the rules and a proof of their correctness.

In the third chapter, we develop a specialization technique based on this transformation system and illustrate its use. We also show how to extend the language of programs in various ways, for example to include conditional definitions and "notations" (such as the *all* notation of Manna and Waldinger [Manna79]). Many sample derivations are presented in this chapter.

In the fourth chapter we give examples of derivations using other transformation methods, such as those of Burstall and Darlington and of Manna and Waldinger. (See [Burstall77] and [Manna79].) These approaches are compared with ours in some detail. The main difference between the systems is that our rules preserve strong equivalence of programs, while the underlying rules of both of these systems do not.

While it is possible to restrict the sequence of application of the Burstall and Darlington rules in such a way that strong equivalence is preserved [Kott78], our approach has the advantage that the rules themselves preserve strong equivalence. Thus, correctness considerations do not impinge on the way our derivations are structured. Program derivations can be a practical

vehicle for the development and documentation of algorithms only if they have a greater intuitive appeal than conventional development techniques and correctness proofs.

The fifth chapter is devoted to an extended example, in which we develop a partial family tree of derivations of context-free parsing algorithms. Recursive versions of Earley's algorithm, the Cocke-Younger-Kasami algorithm, and a top-down algorithm are all derived from the same starting point. These derivations are interesting because the total number of "eureka" steps — those steps requiring significant insight — is quite small.

In the final chapter, we make some general observations on the application of this approach to programming methodology, and we point out some areas deserving further investigation.

Methods for extending our approach to the development of programs with imperative features are outlined in the Appendix.

3. Expression Procedures

We generally make use of two kinds of knowledge in program derivations — properties of the programming language and properties of specific programs defined in that language. These two kinds of facts lead to two kinds of transformation rules on programs. Properties of the programming language form the basis of a generally fixed set of *global* rules for altering the structure of programs, such as the introduction or application of a subroutine. The program-specific facts form the basis of *local* rules for textual modifications, typically the replacement of a portion of program text by new program text that more efficiently computes the same result.

One of the main sources of difficulty in practical program transformation is the development of specialized local rules. Because the rules are represented in a language other than the language of programs, there are no uniform techniques for the introduction, proof, and application of such facts.

We suggest here a new approach to the representation of local properties, which is to give them direct computational meaning in the language of programs. We do this by representing them as a generalized form of procedure, called the *expression procedure*.

This approach has several important consequences: First, we can now manipulate local rules as we would any other part of the program, using the global rules. Second, the set of global transformation rules can be very small. In fact, besides simplification we use only three basic rules, all of which we prove to preserve strong equivalence of programs.

An expression procedure basically represents a method for evaluating a given class of expressions. Consider for example the expression procedure

$$hd(cons(x, y)) \leftarrow x.$$

This definition specifies a method for evaluating expressions that are of the form $hd(cons(x, y))$ where x and y are expressions denoting lists. Now, to evaluate an instance of the expression at run-time an interpreter could, rather than evaluate the constituents of the expression in the usual way, use instead the expression procedure, and directly return a value. If the expression procedure is used then the procedures *hd* or *cons* will not be called.

The expansion of procedure names into their corresponding bodies is a commonly used global transformation. Use of this technique allows us to interpret all procedures, including expression procedures, as transformations: Thus, this rule can be used with the expression procedure above to replace in advance instances of the expression $hd(cons(x, y))$ by the subexpression x.

Expression procedures are different from other procedures in several ways, however. First, their "names" are complex expressions that already have meanings associated with them. This leads us to restrict our attention to expression procedures that are *consistent* — whose left and right hand sides are equivalent *a priori*. Thus, an expression procedure such as

$$hd(cons(x, y)) \leftarrow y$$

is considered *inconsistent*, since the two sides of the definition are not always equivalent. A *consistency requirement is not necessary for conventional pro*cedure definitions, such as

$$f(x) \leftarrow E,$$

for some expression E, since the name 'f' on the left hand side has no meaning except that specified by this definition.

In addition, an expression procedure must represent a ***progressive*** method for computing the expression it defines. The expression procedure,

$$hd(cons(x, y)) \leftarrow hd(cons(x, y))$$

is not progressive given the usual definitions of hd and cons since it introduces looping evaluations of an expression that was previously effectively computable.

Consistency and progressiveness are properties of an expression procedure with respect to some given program. Consider, for example, the program,

$$a \leftarrow 3\,; \qquad\qquad b \leftarrow 3.$$

Either of the expression procedures

$$a + b \leftarrow b + a \qquad\qquad b + a \leftarrow a + b$$

are progressive relative to the given program. After one has been added to the program, however, the other ceases to be progressive, since its addition would introduce looping evaluations where there were none previously.

In general, it may be quite difficult to establish consistency and progressiveness for arbitrary expression procedures. In our application, however, we will introduce expression procedures in a systematic way, using transformation rules that guarantee new expression procedures to be both consistent and progressive.

We can intuitively view an expression procedure as both an assertion and a program. It is a program in that it specifies a method for evaluating certain expressions, yet it is an assertion in that we use it to describe relationships between the names in a program: If all the names appearing in the left hand side of a consistent and progressive expression procedure are defined elsewhere in a program using conventional procedure definitions, then the expression procedure can be eliminated without changing the meaning of that program.

Expression procedures are useful in program transformation because they provide a way of representing program-specific facts directly in the language of programs. Consequently, only a small number of global transformation rules are required for manipulating programs in this extended language.

In the list reversing program derivation, for example, we will use an expression procedure to relate the computations of *rev* and *append*:

$$rev(u) \circ v \leftarrow \textbf{if } u = \Lambda \textbf{ then } v \textbf{ else } rev(tl(u)) \circ cons(hd(u), v).$$

This expression procedure is both consistent and progressive. It is consistent since conventional computations of the left and right hand sides will produce identical results for all u and v. It is progressive, so if conventional evaluation of $rev(u) \circ v$ terminates, then evaluation of $rev(u) \circ v$ using this expression procedure will also terminate.

Note that since the else clause is an instance of the left-hand side, this expression procedure is *recursive*. That is, the expression procedure can be used to evaluate its own else clause, so it can completely compute its result without ever calling *rev* or *append*.

4. Program Manipulation: An Example

Before proceeding with a presentation of the transformation rules, we informally illustrate the use of expression procedures with a simple example of program transformation. (Examples similar to this one appear in [Burstall77] and in [Manna79].) Consider again the inefficient list reversing function,

$$rev(z) \leftarrow \textbf{if } z = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(z)) \circ cons(hd(z), \Lambda).$$

The first step of our derivation will be to develop an expression procedure for the expression

$$rev(u) \circ v$$

where u and v are some expressions. Why do we do this? Since the else clause of *rev* is an instance of this expression, if we can develop an efficient way of computing the expression, then the efficiency of *rev* would be improved. We can consider this phrase to describe a particular *specialized* use of *rev* and *append*, in which the result of *rev* will always be appended to another list.

Starting from the definition of *rev*, it is natural to write the definition,

$$rev(u) \circ v \leftarrow \left(\textbf{if } u = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(u)) \circ cons(hd(u), \Lambda)\right) \circ v.$$

We obtained this by using a transformation rule that allowed us to add '$\circ v$' to both sides of the original definition of *rev*. The use of the transformation rule guarantees that this new expression procedure is consistent and progressive.

In its present state, this expression procedure is not particularly useful, since it produces evaluations for $rev(u) \circ v$ that are equivalent in complexity to conventional evaluations. We can simplify its right hand side, however, taking advantage of the sharing in computation between the two calls to *append*.

We first bring inside the '$\circ v$' on the right hand side of this definition, obtaining

$$rev(u) \circ v \leftarrow \textbf{if } u = \Lambda \textbf{ then } \Lambda \circ v \textbf{ else } [rev(tl(u)) \circ cons(hd(u), \Lambda)] \circ v.$$

We then make use of the three properties of *append*,

$$\begin{gathered}(u \circ v) \circ w = u \circ (v \circ w) \\ cons(u, \Lambda) \circ v = cons(u, v) \\ \Lambda \circ v = v\end{gathered}$$

to simplify this to

$$rev(u) \circ v \leftarrow \textbf{if } u = \Lambda \textbf{ then } v \textbf{ else } rev(tl(u)) \circ cons(hd(u), v).$$

Simplifications such as these are always based on the properties of names we consider to be *primitive* or interpreted; i.e., those names for which there are no explicit procedure definitions. (We discuss the simplification process at length in a later chapter.)

This expression procedure is recursive, as we observed earlier, since the else clause is an instance of the expression, $rev(u) \circ v$. It is also the case that the else clause of the definition of *rev* itself is a call to the expression procedure. Thus, none of the calls to *append* or the recursive calls to *rev* need ever be directly evaluated, since the expression procedure can be used instead. The *cons* calls, on the other hand, must still be computed.

We observe, however, that the number of *cons*'s specified has fortunately turned out to be exactly the minimum number necessary to create a new output list of the appropriate length. Thus, the unmodified basic definition of *rev*, together with this expression procedure as a subroutine, give an optimal

nondestructive list reversal function:

$$rev(x) \leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(x)) \circ cons(hd(x), \Lambda)$$
$$rev(u) \circ v \leftarrow \textbf{if } u = \Lambda \textbf{ then } v \textbf{ else } rev(tl(u)) \circ cons(hd(u), v).$$

It may seem that this gain in efficiency comes only at the great expense of an implemented facility for evaluating expression procedures. Fortunately, this is not the case; it is possible to transform the program further, eliminating the use of the expression procedure in such a way that we can maintain this current level of efficiency.

We do this by abstracting part of the program into a new subroutine. The body of the expression procedure will now consist of a call to this new subroutine, which we will call *rev*2, and the body of *rev*2 will be the former body of the expression procedure.

$$rev(u) \circ v \leftarrow rev2(u, v)$$
$$rev2(u, v) \leftarrow \textbf{if } u = \Lambda \textbf{ then } v \textbf{ else } rev(tl(u)) \circ cons(hd(u), v).$$

It happens that the procedures for *rev* and $rev(u) \circ v$ have matching right hand sides; therefore, we will also replace the body of *rev* with a call to the same subroutine (with appropriate parameters).

$$rev(x) \leftarrow rev2(x, \Lambda)$$

Thus our program now consists of three definitions.

$$rev(x) \leftarrow rev2(x, \Lambda)$$
$$rev(u) \circ v \leftarrow rev2(u, v)$$
$$rev2(u, v) \leftarrow \textbf{if } u = \Lambda \textbf{ then } v \textbf{ else } rev(tl(u)) \circ cons(hd(u), v).$$

Observe that this subroutine formation operation has only introduced an additional layer of definition; it has not changed the order of complexity of the procedures involved.

The expression procedure is now implemented as a set of two mutually recursive subroutines, $rev(u) \circ v$ and *rev*2. By reversing the process that allowed us to form an additional subroutine, we can expand a subroutine call, replacing an instance of a procedure name by its body. We use this rule to

expand the call to the expression procedure in the body of *rev*2, giving the singly recursive procedure,

$$rev2(u, v) \leftarrow \textbf{if } u = \Lambda \textbf{ then } v \textbf{ else } rev2(tl(u), cons(hd(u), v)).$$

Since there are now no more instances of the expression $rev(u) \circ v$ in the program, we know that the expression procedure will never get called, and so it can be eliminated. We are left with the following program,

$$rev(x) \leftarrow rev2(x, \Lambda)$$
$$rev2(u, v) \leftarrow \textbf{if } u = \Lambda \textbf{ then } v \textbf{ else } rev2(tl(u), cons(hd(u), v)).$$

We have thus eliminated the expression procedure while maintaining the efficiency gained through its introduction.

This example illustrates several points about the typical use of expression procedures in program development: Even though we started and ended with sets of definitions that were free of expression procedures, it was the very use of expression procedures to express facts about the relation between the list reversing and list appending functions that gave this derivation its natural character. As we will see, the main use of expression procedures is only in the crucial intermediate steps of derivations such as this; thus, even though the expression procedure is indeed a programming language structure, we will be able to derive the advantages of its use without ever actually developing an implementation for it.

Notice also that throughout this example, the name *rev* continued to denote the same abstract list reversing function, even though its specified method of computation — i.e., its operational meaning — changed significantly because of our transformations. This is because all the transformation rules we use preserve equivalence of programs, as we will prove in the next chapter.

How do we choose when to introduce an expression procedure in a given set of definitions? Observe that in this example, once we introduced the expression procedure, all steps that followed were straightforward and natural. The key insight in the derivation, as in similar *rev* derivations elsewhere, was the choice of the expression $rev(u) \circ v$ for consideration. It is this choice that lets us take advantage of the associative property of *append* to obtain

the simplification. How can we make such choices in general? One possible method is to apply exhaustive search: Since an expression procedure can be used only when it defines an expression that has instances in the given set of definitions, and since there are only finitely many such expressions that may be considered from a given set of definitions, the number of choices is bounded. However, it is of practical importance to find heuristic methods to select useful choices to examine first. This is a difficult question, which we will consider in greater depth later.

5. Programs with Expression Procedures

In the bulk of our investigations, we will be using a simple applicative language of functional recursive programs. Although we define such a language precisely in the next chapter, we outline here some of its characteristics, in order that we may informally introduce the transformation rules.

Syntax. Roughly speaking, a *program* in the language consists of a set of *definitions*, which are expressions of the form

$$A \leftarrow E,$$

where A and E are terms. If the term A, called the *name* part of the definition, is of the form $f(x_1, \ldots, x_n)$, where the x_i are all distinct variables, then the definition is a *basic* definition; otherwise it is called a *complex* definition or *expression procedure*. A *basic* program is thus one that contains only basic definitions. The term E, the *body* of the definition, can contain no variables that do not also occur in A. We often refer to the name part of a complex definition as a *complex name* or *phrase*.

An example of a basic definition is

$$gcd(x, y) \leftarrow \text{if } x \neq 0 \text{ then } gcd(rem(y, x), x) \text{ else } y.$$

Some examples of complex definitions (expression procedures) consistent with this basic definition are

$$\begin{aligned} gcd(x, 2) &\leftarrow \text{if } even(x) \text{ then } 2 \text{ else } 1, \\ gcd(x, x) &\leftarrow x, \\ gcd(x, gcd(y, x)) &\leftarrow gcd(x, y). \end{aligned}$$

In the previous derivation, the *rev* program at one point had the form

$$\begin{aligned} rev(u) \circ v &\leftarrow rev2(u, v) \\ rev(x) &\leftarrow rev2(x, \Lambda) \\ rev2(u, v) &\leftarrow \text{if } u = \Lambda \text{ then } v \text{ else } rev(tl(u)) \circ cons(hd(u), v). \end{aligned}$$

Here, the first definition is an expression procedure, and the second two are both basic definitions.

In any program there must always be certain names that are assumed to be defined elsewhere, since it is impossible to define something from nothing. These externally defined names we call *primitive names*. The other names, the *derivative* names, must be given meaning explicitly by basic definitions.

For example, in the following definition of the factorial function,

$$f(x) \leftarrow \text{if } x = 0 \text{ then } 1 \text{ else } x \cdot f(x-1),$$

the name f is the sole derivative name, and **if then else**, $=$, 0, etc., are primitive names defined over suitable domains. In the *rev* program above, all names are primitive except *rev* and *rev2*.

We impose the restriction that there must be exactly one basic definition for each derivative name in a given program. This implies that all names occurring in expression procedure names and bodies are either primitive or have basic definitions; thus, the consistency property of an expression procedure can be tested with respect to the basic part of the program in which it occurs.

Semantics. We give meaning to programs in this language of definitions by means of an *evaluator* model. In such a model, we assume that all primitive names are interpreted, in the sense that their meanings are built into the simplification mechanism of an evaluator. This reflects our intuitive associations of the primitive names and constants of a language with the basic instructions and data values of a machine. The derivative names, those names defined by the programmer, are given meaning explicitly by the evaluation process:

To compute the meaning of some expression, we *evaluate* it, replacing some derivative names or phrases by the bodies of their definitions, simplifying, and then repeating the same process for the resulting expression. Evaluation terminates when only primitive names remain. Of course, for some

sets of definitions, evaluation may never terminate. Thus in this operational sense, definitions correspond to procedures.

Observe that in the presence of expression procedures there is, in general, more than one way to evaluate a given name. When an expression defined by an expression procedure is encountered, it can be evaluated either conventionally or by using the expression procedure. If the expression procedure is consistent and progressive, however, then the results of both evaluations will be identical, and if one evaluation terminates than so will the other. Thus, although evaluation can be nondeterministic, all choices will eventually lead to the same result. When we speak informally of the *complexity* of a program, we mean the length of the shortest possible of these evaluations.

Besides simplification, there are four fundamental program transformation rules we use. Two of these introduce new definitions: *Abstraction* introduces new basic definitions and *composition* introduces new expression procedures. The third rule, *application*, expands both types of definitions, replacing definition names by definition bodies. The last rule allows elimination of unnecessary definitions.

We now describe these four rules in an informal manner. Although the rules are stated precisely and proved correct in the second chapter, the description here is sufficient for understanding the examples in Chapter 3.

6. The Abstraction Transformation

Naming is a fundamental operation for introducing structure in a set of definitions. We consider now a rule for defining new basic names. This *abstraction* operation corresponds intuitively to the familiar operation of forming a new subroutine for an existing section of code in a program.

We use the notation $E[F]$ to indicate that an expression E contains one or more instances of some subexpression F (in referentially transparent contexts — contexts in which F can be replaced by any expression that evaluates equivalently).

The idea of abstraction is to assign a name to some subexpression that occurs one or more times in a given definition (or definitions), and then to

use this new name in place of the subexpression. That is, if N is a new basic name, the result of abstracting F from $E[F]$ in the definition $A \leftarrow E[F]$ is the pair of definitions,

$$A \leftarrow E[N], \quad N \leftarrow F.$$

For example, from the definition

MOON ← *the largest natural satellite of the third planet*

we can abstract the phrase, *the third planet*, by creating a new name EARTH. The following pair of definitions replace the given one:

MOON ← *the largest natural satellite of* EARTH
EARTH ← *the third planet.*

The abstraction rule can be written schematically as

replace $A \leftarrow E[F]$ **by** $A \leftarrow E[N], \quad N \leftarrow F.$

The other definitions in the program are unaffected by this operation.

More generally, a given expression can be simultaneously abstracted from several definitions in a set of definitions. This is what we did in the *rev* example when we formed the new definition for *rev*2. We replaced the pair of definitions,

$$rev(x) \leftarrow \text{if } x = \Lambda \text{ then } \Lambda \text{ else } rev(tl(x)) \circ cons(hd(x), \Lambda)$$
$$rev(u) \circ v \leftarrow \text{if } u = \Lambda \text{ then } v \text{ else } rev(tl(u)) \circ cons(hd(u), v)$$

by the following set of three definitions,

$$rev(x) \leftarrow rev2(x, \Lambda)$$
$$rev(u) \circ v \leftarrow rev2(u, v)$$
$$rev2(u, v) \leftarrow \text{if } u = \Lambda \text{ then } v \text{ else } rev(tl(u)) \circ cons(hd(u), v).$$

by abstracting the right hand sides of both into a new subroutine, *rev*2, and introducing appropriate parameters. (Because of our use of parameters,

an abstracted expression can be a generalization of the expression before abstraction.)

This more general version of abstraction can be written

$$\begin{aligned}&\textbf{replace } A_1 \leftarrow E_1[F], \quad \ldots, \quad A_k \leftarrow E_k[F]\\ &\quad\textbf{by } A_1 \leftarrow E_1[N], \quad \ldots, \quad A_k \leftarrow E_k[N], \quad N \leftarrow F\end{aligned}$$

We require that the new name N have sufficient parameters to capture all the free variables in the abstracted expression F; this guarantees that there will no free variables in the new definition of N. Furthermore, for certain evaluator models we may have to restrict the range of legal choices for F in order to ensure that termination properties will be preserved. (We will discuss both of these issues in the next chapter.)

One use of abstraction is in the elimination of redundant subcomputations (i.e., common subexpression elimination). Consider for example the *last* function, which computes the last element of a list x:

$$\begin{aligned}last(x) \leftarrow\ &\textbf{if } x = \Lambda \textbf{ then } \Lambda\\ &\textbf{elseif } tl(x) = \Lambda \textbf{ then } hd(x)\\ &\textbf{else } last(tl(x)).\end{aligned}$$

Observe that $tl(x)$ will be computed twice in all but one iteration of this program. Abstraction can be used to eliminate this extra computation.

One possible way of abstracting the elseif part of the body gives

$$\begin{aligned}last(x) &\leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } lasta(tl(x), hd(x))\\ lasta(u, v) &\leftarrow \textbf{if } u = \Lambda \textbf{ then } v \textbf{ else } last(u).\end{aligned}$$

Here we abstracted most of the elseif clause of the original definition into a new function, *lasta*; however, the computations of $tl(x)$ and $hd(x)$ have been retained in the main definition. Thus, in a call-by-value evaluator, $tl(x)$ and $hd(x)$ are both computed exactly once for each call to *lasta*. Alternatively, we could have abstracted to obtain

$$\begin{aligned}last(x) &\leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } lastb(tl(x), x)\\ lastb(u, v) &\leftarrow \textbf{if } u = \Lambda \textbf{ then } hd(v) \textbf{ else } last(u).\end{aligned}$$

In this case, the computation of *hd* was left in the auxiliary function. This has the advantage that both *tl* and *hd* will never be computed more than necessary. In the previous version *hd* was computed on every iteration, but was only needed on the last.

We could also have abstracted to obtain

$$\begin{aligned}last(x) &\leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } lastc(x)\\ lastc(u) &\leftarrow \textbf{if } tl(u) = \Lambda \textbf{ then } hd(u) \textbf{ else } last(tl(u)),\end{aligned}$$

but this does not improve efficiency as *lasta* or *lastb* did. Another useless but correct abstraction yields,

$$\begin{aligned}last(x) &\leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } lastd(x, tl(x), tl(x), \Lambda)\\ lastd(x, u, v, w) &\leftarrow \textbf{if } u = w \textbf{ then } hd(x) \textbf{ else } last(v).\end{aligned}$$

This illustrates that constants and multiple subexpression instances can all be abstracted separately.

As a final *last* example, consider the following attempt at abstraction:

$$\begin{aligned}last(x) &\leftarrow laste(x, tl(x))\\ laste(u, v) &\leftarrow \textbf{if } u = \Lambda \textbf{ then } \Lambda \textbf{ elseif } v = \Lambda \textbf{ then } hd(u) \textbf{ else } last(v).\end{aligned}$$

If this program is evaluated using the conventional call-by-value technique, then it will always produce an error, since at some point $tl(\Lambda)$ will be computed. We will see that for functional programs the abstraction rule is defined to avoid such cases, thus maintaining the equivalence preserving property of the rule. (Actually, the correctness of this particular abstraction depends on our convention for handling errors. We will consider the various choices in a later section.)

7. The Application Transformation

The abstraction rule provides us with a way of introducing new basic definitions into a program. A second transformation rule is needed for expanding definitions — replacing derivative names by their corresponding definition bodies.

By expanding an instance of a name in an expression, we can simplify its associated body in the particular specialized context the name appears in. The *application* or *unfolding* rule provides a means for doing this. Application is used to replace an instance of a basic or complex name by the body of its definition. In schematic form, we have

$$\text{given} \quad N \leftarrow F\,, \qquad \text{replace } A \leftarrow E[N] \quad \text{by} \quad A \leftarrow E[F].$$

Any arguments of N in $E[N]$ must be substituted for occurrences of the corresponding parameters in the substituted instances of F. For example, we could unfold the instance of the basic name *rev* in the body of the definition,

$$rev(x) \leftarrow \text{if } x = \Lambda \text{ then } \Lambda \text{ else } rev(tl(x)) \circ cons(hd(x), \Lambda),$$

to get the equivalent definition

$$\begin{aligned} rev(x) \leftarrow \text{if } x = \Lambda \text{ then } \Lambda \\ \text{else } \big(\text{if } tl(x) = \Lambda \text{ then } \Lambda \\ \text{else } rev(tl(tl(x))) \circ cons(hd(tl(x)), \Lambda)\big) \\ \circ cons(hd(x), \Lambda). \end{aligned}$$

Here, we substituted the argument, $tl(x)$, for every instance of x in the substituted definition body.

Application can be used to make simple changes to the structure of recursions, such as the transformation of a simple mutual recursion into a single recursion. For example, at one point in the *rev* derivation, our program included the mutually recursive pair,

$$\begin{aligned} rev(u) \circ v &\leftarrow rev2(u, v) \\ rev2(u, v) &\leftarrow \text{if } u = \Lambda \text{ then } v \text{ else } rev(tl(u)) \circ cons(hd(u), v). \end{aligned}$$

The application rule allows us to unfold the call to the expression procedure $rev(u) \circ v$ in the body of *rev2*, giving

$$rev2(u, v) \leftarrow \text{if } u = \Lambda \text{ then } v \text{ else } rev2(tl(u), cons(hd(u), v)).$$

Application serves the function of a symbolic evaluation rule. For example, let f be

$$f \leftarrow tl(rev(cons(a, \Lambda))),$$

where *rev* is as defined above and a is any value. The application rule can be used to determine the value of f. We can apply the definition of *rev* in the body of f to obtain

$$\begin{aligned} f \leftarrow tl\big(&\text{if } cons(a, \Lambda) = \Lambda \text{ then } \Lambda \\ &\text{else } rev(tl(cons(a, \Lambda))) \circ cons(hd(cons(a, \Lambda)), \Lambda)\big). \end{aligned}$$

Using the properties

$$\begin{aligned} cons(a, b) &\neq \Lambda \\ hd(cons(a, b)) &= a \\ tl(cons(a, b)) &= b, \end{aligned}$$

and the definition of **if then else**, this simplifies to

$$f \leftarrow tl(rev(\Lambda) \circ cons(a, \Lambda)),$$

Applying *rev* again, and simplifying, we obtain,

$$f \leftarrow tl(\Lambda \circ cons(a, \Lambda)).$$

Since *append* is primitive, we can simplify this to

$$f \leftarrow \Lambda.$$

This example illustrates the strong parallel between application and the evaluation operation described earlier. Unfolding a definition and simplifying the result is analagous to doing one step of evaluation; in fact, a trace of the evaluation of the original definition of f would exactly mirror this transformation sequence. Work is thus transferred from "run time," when it would have to be done every time f is evaluated, to "development time," when it is done exactly once, no matter how many times the new definition of f will get evaluated.

The notion of application or unfolding is a familiar one in computer science. Fixed-point invariant transformations, symbolic evaluation, β-conversion, Burstall and Darlington's *unfold* transformation, and Wegbreit's *partial application* are all examples of application rules. (See e.g., [Manna74], [Cheatham79], [Wegbreit75], [Boyer75], and [Burstall77].)

8. The Composition Transformation

The abstraction rule is used to introduce new basic definitions into a set of definitions. In a similar way, the *composition* rule provides a syntactic means for introducing new expression procedures.

The expression procedures introduced by the composition rule are always consistent and progressive.

The *composition* rule may be stated as follows: Let $A \leftarrow F$ be a definition, and E be an expression. Then in schematic form, we have

$$\text{given} \quad A \leftarrow F, \qquad \text{add } E[A] \leftarrow E[F].$$

That is, we can embed a definition $A \leftarrow F$ in an expression E to create a new expression procedure for the expression $E[A]$. (In the precise statement of this rule, certain restrictions are imposed on the form of E.)

The composition rule is a generalization of the partial application operation. (See e.g., [Burstall71] and [Haraldsson77].)

For example, suppose we had the two definitions of sets,

$$\begin{aligned} \text{PRIMES} &\leftarrow P \\ \text{EVENS} &\leftarrow V, \end{aligned}$$

where P and V are expressions that describe the sets of prime and even numbers respectively. One way we could form an expression procedure for PRIMES $\cap$ EVENS is to compose the first of these definitions inside the expression

$$E[A] = A \cap \text{EVENS},$$

to get the expression procedure,

$$\text{PRIMES} \cap \text{EVENS} \leftarrow P \cap \text{EVENS}.$$

The application rule then enables us to unfold the instance of EVENS in the body of this definition, yielding,

$$\text{PRIMES} \cap \text{EVENS} \leftarrow P \cap V.$$

Finally, some significant amount of simplification would reduce the expression $P \cap V$ to $\{2\}$, yielding the final definition,

$$\text{PRIMES} \cap \text{EVENS} \leftarrow \{2\}.$$

In the previous *rev* example, we used the composition rule in the very first step of the derivation to introduce the expression procedure for $rev(u) \circ v$. We embedded the definition,

$$rev(z) \leftarrow \textbf{if } z = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(z)) \circ cons(hd(z), \Lambda)$$

in a simple expression involving the *append* function,

$$E[A(u), v] = A(u) \circ v,$$

to get the phrase definition

$$rev(u) \circ v \leftarrow [\textbf{if } u = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(u)) \circ cons(hd(u), \Lambda)] \circ v.$$

Alternatively we could have produced the expression procedure,

$$u \circ rev(v) \leftarrow u \circ [\textbf{if } v = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(v)) \circ cons(hd(v), \Lambda)],$$

but this would not be very useful, since there are no instances of the phrase it defines anywhere in the program.

Observe that composition can be used to instantiate definitions. For example, we could compose the definition of *rev* in the expression,

$$E[A(u)] = A(tl(u)),$$

giving

$$rev(tl(u)) \leftarrow \textbf{if } tl(u) = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(tl(u))) \circ cons(hd(tl(u)), \Lambda),$$

which is recursive.

As a final example, we could have composed *rev* in the complicated expression,

$$E[A(u), v] = A(tl(u)) \circ cons(hd(u), v),$$

to obtain

$$rev(tl(u)) \circ cons(hd(u), v)$$
$$\leftarrow \big(\text{if } tl(u) = \Lambda \text{ then } \Lambda$$
$$\text{else } rev(tl(tl(u))) \circ cons(hd(tl(u)), \Lambda)\big)$$
$$\circ\, cons(hd(u), v),$$

which simplifies to

$$rev(tl(u)) \circ cons(hd(u), v)$$
$$\leftarrow \text{if } tl(u) = \Lambda \text{ then } cons(hd(u), v)$$
$$\text{else } rev(tl(tl(u))) \circ cons(hd(tl(u)), cons(hd(u), v)),$$

and is thus recursive. This expression procedure could have been used in the *rev* derivation instead of $rev(u) \circ v$, and the final result would have been the program

$$rev(x) \leftarrow \text{if } x = \Lambda \text{ then } \Lambda \text{ else } revx(x, \Lambda)$$
$$revx(u, v) \leftarrow \text{if } tl(u) = \Lambda \text{ then } cons(hd(u), v)$$
$$\text{else } revx(tl(u), cons(hd(u), v)),$$

which is more complicated than, but as efficient as the reverse program we derived earlier.

9. The Definition Elimination Transformation

The last of the four transformations is used to eliminate unwanted or unnecessary definitions from a program. In particular, any expression procedure can be eliminated, and any basic definition which is not referenced can be eliminated.

We used this transformation in the *rev* derivation to eliminate the expression procedure for $rev(u) \circ v$ when it was no longer referenced.

Chapter 2

The Transformations and their Correctness

In this technical chapter, we specify the meaning of functional programs with expression procedures, and state and prove the program transformation rules.

The meaning of programs is specified operationally, by means of a simple *call-by-value* evaluator model. The call-by-value evaluation scheme corresponds most closely with existing programming language implementations. The transformation rules are also stated, but not proved, for a *call-by-name* evaluation model.

Manna's *book*, [Manna74], contains sufficient background material for this chapter. The theses of Cadiou and Vuillemin discuss some issues related to the evaluator models presented here [Cadiou72], [Vuillemin73]. Our full evaluator model is similar to the evaluator that is implemented in Weyhrauch's FOL system [Weyhrauch79].

This chapter can be omitted on first reading of the thesis.

1. Syntax of Programs

Our programming language is a simple language of recursive functional programs.

We assume the existence of disjoint sets of *constant* symbols, *variable* symbols, *derivative* symbols, and *primitive* symbols.

The set of *atomic terms* is the union of the sets of

(a) *constant* symbols c, d, ..., and

(b) *variable* symbols x, y, ...

The set of *terms* is defined by the following rules:

(a) An atomic term is a term.

(b) If f^n is an n-ary *derivative* symbol and t_1, t_2, ..., t_n are terms, then $f^n(t_1, t_2, \ldots, t_n)$ is a term.

(c) If g^n is an n-ary *primitive* symbol and t_1, t_2, ..., t_n are terms, then $g^n(t_1, t_2, \ldots, t_n)$ is a term.

The meta-variables s, t, u, ... range over terms, f, h, k; ... range over derivative symbols, and g, g', ... range over primitive symbols.

Primitive and derivative symbols are *function* symbols. A term is *primitive* if it contains no derivative symbols.

Examples of terms are 5, x, $f(x, y)$, and $g(h(x), 3)$. Often, an infix or other syntax may be used for terms, as in $(x * y) + 5$ or **if** P **then** x **else** y. Observe that conditionals are considered primitive functions in this treatment.

Though we do not do it here, it is generally convenient to assume that variable and constant symbols are assigned types.

The function *vars* mapping terms to sets of variable symbols is defined inductively as follows:

$$\begin{aligned} &vars[\![f(t_1, \ldots, t_n)]\!] = vars[\![t_1]\!] \cup \ldots \cup vars[\![t_n]\!] \\ &vars[\![g(t_1, \ldots, t_n)]\!] = vars[\![t_1]\!] \cup \ldots \cup vars[\![t_n]\!] \\ &vars[\![x_i]\!] = \{x_i\} \\ &vars[\![c_i]\!] = \emptyset. \end{aligned}$$

For example, $vars[\![k(x, f(x, y), 3)]\!] = \{x, y\}$.

A term t is *ground* if $vars[\![t]\!] = \emptyset$. For example, the term **if** $p(2)$ **then** 1 **else** $2 \cdot f(2 - 1)$ is a ground term.

A *definition* is an expression of the form

$$t_1 \leftarrow t_2,$$

where t_1 is a nonatomic term and t_2 is any term. The left-hand term t_1 is called the *name* part of the definition, and t_2 is called the *body* part.

A *basic term* is a term of the form $f^n(x_1, x_2, \ldots, x_n)$ where the x_i are variable symbols and $i \neq j$ implies x_i and x_j are distinct. All other nonatomic terms are called *complex* terms or *phrases*. A definition is *basic* if its name part is a basic term. All other definitions are *complex* definitions or *expression procedures*.

Examples of definitions are

$$\begin{gathered} f(x) \leftarrow 3, \\ h(x, y) \leftarrow \textbf{if } P(x) \textbf{ then } k(y + 1) \textbf{ else } k(y - 1), \\ k(x, z) \leftarrow x + y + g(z), \\ f(h(x, z)) \leftarrow g(h(x)), \text{ and} \\ rev(u) \circ v \leftarrow \textbf{if } u = \Lambda \textbf{ then } v \textbf{ else } rev(tl(u)) \circ cons(hd(u), v). \end{gathered}$$

The first three of these are basic; the other two are complex.

A definition $t_1 \leftarrow t_2$ is *well-formed* if $vars[\![t_1]\!] \supseteq vars[\![t_2]\!]$. Thus, the definition $f(x, y) \leftarrow g(x, h(y, z))$ is not well-formed, but $f(x, y, z) \leftarrow g(h(x), y)$ is.

A *program* is an expression of the form

$$f_1, \ldots, f_m : \ d_1; \ldots; d_n$$

where the f_i are derivative symbols and d_j are definitions. The f_i's are called the *principal symbols* of the program. The function symbols in d_1, ..., d_n that are not principal are called *local symbols*.

Local and principal symbols are distinguished to allow the introduction and elimination of local auxiliary definitions while maintaining equivalence of programs. That is, a definition that is not referenced within a program but that is referenced from the outside (i.e., is a principal symbol) cannot be eliminated.

A *basic program* is a program all of whose definitions are basic. The *basic segment* of a program is that basic program consisting of the principal symbols and the basic definitions of P.

A term t is a *legal input* for a program P if it is ground and if all its function symbols are either primitive or principal.

A program is *well-formed* if

(a) all of its definitions are well-formed, and

(b) every derivative name that occurs in the name or body of a definition or is principal is defined by exactly one basic definition.

For example, the program

$$\begin{aligned} rev: \quad & rev(x) \leftarrow rev2(x, \Lambda)\ ; \\ & rev(u) \circ v \leftarrow rev2(u, v)\ ; \\ & rev2(u, v) \leftarrow \text{if } u = \Lambda \text{ then } v \text{ else } rev(tl(u)) \circ cons(hd(u), v) \end{aligned}$$

is well-formed. The first and third definitions are basic; the second is an expression procedure. The derivative symbols are *rev* and *rev2*; *rev* is principal, and *rev2* is local.

Unless otherwise stated, all definitions and programs are assumed to be well-formed. In addition, the principal symbols specification will be omitted in those examples in which the set of principal names is obvious from context.

2. Substitutions

A *substitution pair* is a pair $[x \leftarrow t]$, where x is a variable and t is a term. A *substitution* is a finite set of substitution pairs $\{[x_1 \leftarrow t_1], \ldots, [x_n \leftarrow t_n]\}$ such that $i \neq j$ implies that the variables x_i and x_j are distinct.

If t is a term and θ is a substitution, then the instantiation function $t \circ \theta$ is defined inductively as follows:

$$\begin{aligned} & f(t_1, \ldots, t_n) \circ \theta = f(t_1 \circ \theta, \ldots, t_n \circ \theta) \\ & g(t_1, \ldots, t_n) \circ \theta = g(t_1 \circ \theta, \ldots, t_n \circ \theta) \\ & x \circ \theta = \begin{cases} t', & \text{if } [x \leftarrow t'] \in \theta\text{:} \\ x, & \text{otherwise.} \end{cases} \\ & c \circ \theta = c \end{aligned}$$

It is often more convenient to use a different notation for instantiation:

$$t'[u_1, \ldots, u_n] \text{ for } x_1, \ldots, x_n \ = \ t' \circ \{[x_1 \leftarrow u_1], \ldots, [x_n \leftarrow u_n]\}.$$

The *for* specifications may be omitted when obvious from context. In this notation, the substitution specifications associate to the left; thus, we may write $r[s][u]$ for $(r[s])[u]$. If

$$t \ = \ t'[u_1, \ldots, u_n] \text{ for } x_1, \ldots, x_n,$$

for some $u_1, \ldots, u_n$ and $x_1, \ldots, x_n$, then the term t is an *instance* of the term t'.

Note that these substitutions are parallel substitutions. Thus,

$$\begin{aligned} f(x, g(y), z)[y, x] \text{ for } x, y \ &= \ f(x, g(y), z) \circ \{[x \leftarrow y], [y \leftarrow x]\} \\ &= \ f(y, g(x), z). \end{aligned}$$

Often, rather than writing a specification for a list of variables or terms like "$u_1, \ldots, u_n$," we may instead write simply "$\overline{u}$." Thus, we could have written $t = t'[\overline{u}]$ *for* $\overline{x}$ for the expression above.

Frequently, we will use this notation as a means of distinguishing specific subterms of a given term. That is, for a given term t, we may say $t = t'[f(\overline{u})]$ to distinguish the particular subterm, $f(\overline{u})$, of t.

3. Simplifications

An evaluator model assigns meaning to the derivative symbols of a program in terms of the meanings of a given set of primitive symbols. The meanings of the primitive symbols are specified to the evaluator using primitive simplification rules.

A *primitive simplification rule* is an expression of the form

$$g(c_1, \ldots, c_i, *_{i+1}, \ldots, *_n) \rightarrow a$$

where c_j are constants, $*_j$ are *pattern symbols*, i is less than or equal to n, and a stands for either a constant symbol or a pattern symbol $*_k$ where $i < k \leq n$.

A primitive simplification rule $g(c_1, \ldots, c_i, *_{i+1}, \ldots, *_n) \rightarrow a$ ***matches*** a term $g(t_1, \ldots, t_n)$ if $t_1 = c_1, \ldots, t_i = c_i$. If a is a constant symbol c then the ***result*** of matching is c; otherwise a is $*_k$, and the ***result*** is the term t_k.

The notion of primitive simplification rule essentially reflects the machine implementation of sequential base functions for operations such as arithmetic and transfer of control. Multiplication, for example, would be specified by a set of rules like $mult(3, 2) \rightarrow 6$. The pattern variables allow for rules that do not require all of their arguments to be evaluated, as in

$$if(\textbf{true}, *_2, *_3) \rightarrow *_2 \qquad if(\textbf{false}, *_2, *_3) \rightarrow *_3.$$

Note that the syntax does not allow for a third rule such as

$$if(*_1, c, c) \rightarrow c,$$

to be added, unless the arguments to the *if* are reordered. But if the arguments are reordered, then the first two rules would no longer be acceptable. This restriction on pattern variables essentially guarantees that if arguments to primitive functions are evaluated sequentially, from left to right, then simplification rules will be used as soon as they are applicable, with no unnecessary evaluation.

If the definitions above are altered so that pattern symbols and constants can be freely mixed, then parallel evaluation or a simulation thereof is required for the simplifications to be promptly applied. (This notion can be made more precise, though we do not do it here.) Such an evaluator would allow, for example, a multiplication operation that has the two rules,

$$mult(0, *_2) \rightarrow 0 \qquad mult(*_1, 0) \rightarrow 0.$$

A set S of primitive simplification rules is ***well-formed*** if for each primitive symbol g, exactly one rule matches each term of the form $g(c_1, \ldots, c_n)$. Note that this implies that at most one rule matches every term $g(t_1, \ldots, t_n)$.

These restrictions unfortunately require us to give meaning to such constructions as $plus(\textbf{true}, 3)$; this problem could be circumvented, however, in a language in which atomic terms are assigned types.

Unless otherwise stated, all sets of primitive simplification rules are assumed to be well-formed.

Let S be well-formed. The function $simprule_S[\![t]\!]$ is defined such that

(a) if there exists a rule in S that matches t, then $simprule_S[\![t]\!]$ is the result of matching;

(b) otherwise, $simprule_S[\![t]\!] = t$.

(We omit the subscript when it is obvious from context.)

For example, $simprule[\![if(\textbf{true}, t_1, t_2)]\!] = t_1$ for any terms t_1 and t_2.

Given a well-formed set of primitive simplification rules, S, the function $simp_S$ mapping terms to terms is defined inductively as follows:

$$\begin{aligned} &simp[\![f(t_1, \ldots, t_n)]\!] = f(simp[\![t_1]\!], \ldots, simp[\![t_n]\!]) \\ &simp[\![g(t_1, \ldots, t_n)]\!] = simprule_S[\![g(simp[\![t_1]\!], \ldots, simp[\![t_n]\!])]\!] \\ &simp[\![x_i]\!] = x_i \\ &simp[\![c_i]\!] = c_i. \end{aligned}$$

Observe that the result of applying *simp* to a primitive ground term is always a constant symbol.

Thus, given appropriate simplification rules for *if* and *plus*,

$$simp[\![plus(2, if(\textbf{false}, f(3), 5)]\!] = 7,$$

for any function symbol f.

The primitive simplification ***transformations*** that are used in program derivations, such as commutativity or associativity of addition, are all derived from the appropriate primitive simplification rules. Some examples of such transformations are

$$\begin{aligned} hd(cons(a, b)) &\Rightarrow a \\ m + n &\Rightarrow n + m \\ \textbf{if } s \textbf{ then true else false} &\Rightarrow s \\ \textbf{if true then } s \textbf{ else } t &\Rightarrow s, \end{aligned}$$

where a and b are terms that evaluate to lists, m and n are terms that evaluate to nonnegative integers, and s and t are any terms. The justification for

such transformations requires reasoning over appropriate sets of primitive simplification rules. For example, the commutativity rule above can only be introduced after it has been established that for all constants m and n denoting nonnegative integers, $simp[\![m+n]\!] = simp[\![n+m]\!]$. The last rule, on the other hand, follows directly from the corresponding primitive simplification rule for conditionals. We will make implicit use of transformations such as these in our examples.

4. Strictness

Let S be a well-formed set of primitive simplification rules. A primitive symbol g is *fully strict* in S if there are no pattern symbols in any of the rules for g in the set S. The primitive symbol g is *strict in the i^{th} position* if the pattern symbol $*_i$ does not occur in any of the rules for g in S. For example, *cons* and *plus* are fully strict, while *if* is strict only in the first position.

Note that if g is strict in the i^{th} position, and if $i > 1$, then it must also be strict in the $(i-1)^{st}$ position. Note also that if g is fully strict in S then the value of $simprule[\![g(t_1,\ldots,t_n)]\!]$ is a constant symbol if and only if each of the terms t_i is a constant symbol.

Let t be a term and S be a set of primitive simplification rules. The *strict subterms* of t are defined inductively as follows:

(a) The term t is a strict subterm of t.

(b) If $f(t_1,\ldots,t_n)$ is a strict subterm of t, then so are each of the terms t_i.

(c) If $g(t_1,\ldots,t_n)$ is a strict subterm of t, then for every i such that g is strict in the i^{th} position, the term t_i is a strict subterm of t.

Thus, the strict subterms of

$$f(if(g(x),h_1(z),h_2(z)),\ plus(h_3(h_4(y)),z))$$

are $g(x)$, $if(g(x),h_1(z),h_2(z))$, y, $h_4(y)$, $h_3(h_4(y))$, z, $plus(h_3(h_4(y)),z)$, the term itself, and x if g is strict in the first position.

Note that a term t' can be a strict subterm of a term t even if there is an instance of t' that is not in a strict position.

We will prove below that all the strict subterms of a term t are fully expanded in a call-by-value evaluation of t. Thus, the notion of strict subterm gives a syntactic characterization of a useful subset of the set of terms that must be expanded by a call-by-value evaluation algorithm.

5. Evaluation for Basic Programs

We consider here only call-by-value evaluation, although the results can easily be extended to other evaluation rules.

Given a term t, a subterm t_1 is *to the left of* a subterm t_2 if there exists a subterm $h(\ldots,t_1',\ldots,t_2',\ldots)$ of t such that t_1 is a subterm of t_1' and t_2 is a subterm of t_2'. A term is a *leftmost* term if there are no nonprimitive terms to the left of that term. For example, the leftmost-innermost choice in the term $fib(fib(2)+fib(3))$ is $fib(2)$.

Given a basic program P and a set of primitive simplification rules S, the procedure $Evb_P[\![t]\!]$ on legal input terms is defined as follows:

$Evb_P[\![t]\!]$:

(1) (*Simplify*) Let t be $simp[\![t]\!]$. (t is called the *current term.*)

(2) If t is now primitive, then exit the evaluator with t as the result. (Otherwise, t contains a derivative name.)

(3) (*Choose*) Since the program P is well-formed, there is a unique definition $f(\overline{x}) \leftarrow s$ such that there is a term t',

$$t = t'[f(\overline{u})] \text{ for } z,$$

where each of the terms u_i is primitive and z occurs exactly once in t', and in a leftmost position. (f is a *leftmost-innermost* choice.)

(4) (*Expand*) Let t be $t'[s[\overline{u}] \text{ for } \overline{x}] \text{ for } z$.

(5) Go to step (1). ∎

Observe that if this evaluation procedure terminates, its result will be a constant symbol. Observe also that there is exactly one leftmost-innermost

choice for a given nonprimitive current term. Thus, this evaluator is deterministic.

This substitution model of evaluation is equivalent to the more usual recursive evaluator model for call-by-value, in which function arguments are evaluated separately from the function itself. We use the substitution model because it generalizes to programs with expression procedures in a natural way.

We say $Evb_{P_1}[\![t_1]\!] = Evb_{P_2}[\![t_2]\!]$ if

(a) (*Co-termination*) $Evb_{P_1}[\![t_1]\!]$ terminates if and only if $Evb_{P_2}[\![t_2]\!]$ terminates, and

(b) (*Weak Equivalence*) when they both terminate, they have the same result.

Two programs are *equivalent* if they have the same set of legal input terms and if, for every legal input term t, $Evb_{P_1}[\![t]\!] = Evb_{P_2}[\![t]\!]$.

An *evaluation sequence* for a term t is the possibly infinite sequence of current terms in the evaluation of t.

As an example of basic evaluation, consider the Fibonacci function over the nonnegative integers,

$$fib(x) \leftarrow \textbf{if } x \leq 1 \textbf{ then } x \textbf{ else } fib(x-1) + fib(x-2).$$

Suppose the input term at step (1) of Evb is

$$fib(fib(2) + fib(3)).$$

This term simplifies to itself; so the current term t at step (3) will also be $fib(fib(2) + fib(3))$. Therefore, t' will be $fib(z + fib(3))$ and u will be 2. In step (4), $s[u]$ will be the body of fib with z instantiated to 2, so t becomes $t'[s[u]]$, which is

$$fib\big(\textbf{if } 2 \leq 1 \textbf{ then } 2 \textbf{ else } fib(2-1) + fib(2-2) \;+\; fib(3)\big).$$

After simplification in step (1), the new current term will be

$$fib\big(fib(1) + fib(0) \;+\; fib(3)\big).$$

The evaluation sequence for $fib(2)$ is:

$fib(2)$	**Given**
if $2 \leq 1$ **then** 2 **else** $fib(2-1) + fib(2-2)$	**Expand**
$fib(1) + fib(0)$	**(Simplify)**
$\big($**if** $1 \leq 1$ **then** 1 **else** $fib(1-1) + fib(1-2)\big) + fib(0)$	**Expand**
$1 + fib(0)$	**(Simplify)**
$1 + \big($**if** $0 \leq 1$ **then** 0 **else** $fib(0-1) + fib(0-2)\big)$	**Expand**
1	**(Simplify)**

We now state three rather obvious but useful lemmas concerning evaluation. The following simple examples illustrate the sense of these lemmas: Lemma 1 establishes that in evaluation of the term $fib(fib(a)+fib(b))$, all three instances of fib will be expanded. Lemma 2 proves that in this term, $fib(a)$ will be completely evaluated before evaluation proceeds into $fib(b)$. Lemma 3 establishes that if $fib(a) - fib(b) = fib(c)$ then

$$fib(fib(a) - fib(b)) = fib(fib(c)).$$

Lemma 1 (Strictness Lemma): If evaluation of a ground term t terminates, then so must evaluation of every strict subterm of t.

Proof. The proof is by induction on the structure of terms.

Base case. The term t is a variable x. This case cannot occur since a ground term contains no variables.

Base case. The term t is a constant c. This case holds trivially.

Induction step. The term t is of the form $f(t_1, \ldots, t_n)$. The strict subterms of t are the terms t_i, and their strict subterms. Since evaluation of t terminates, the definition of f must eventually be expanded. By the definition of leftmost-innermost choice, this can only happen after all the terms t_i have been evaluated. The only other strict subterms of t are the strict subterms of these terms, which must be evaluated by the induction hypothesis.

Induction step. The term t is of the form $g(t_1, \ldots, t_n)$. The strict subterms of t are the terms t_i that occur in strict positions of g, and their

strict subterms. Since evaluation of t terminates, t must eventually simplify to a constant. Before any simplification rule can apply to g, however, the terms t_i in strict positions must be completely evaluated. The only other strict subterms of t are the strict subterms of these terms, which must be evaluated by the induction hypothesis. ∎

Lemma 2: Let $t[u]$ *for* x be a current term in *Evb* such that x occurs exactly once in t. If any derivative symbol in the subterm u is expanded by *Evb*, then all of u will be expanded before any derivative symbols outside u are.

Proof. If *Evb* chooses a derivative symbol in u, then u is leftmost in the current term. By the definition of leftmost, the only leftmost terms that are not subterms of u include u as a subterm. But the arguments of these terms cannot all be primitive until u is fully expanded, so the only possible leftmost-innermost choices are in u. ∎

Corollary: Let $t = u[r]$ be a ground term whose evaluation terminates in the two equivalent basic programs B_1 and B_2. Then a subtree for r occurs in the evaluation tree for t in B_1 if and only if one occurs in the tree for B_2.

Proof. Suppose a subtree for r occurs for B_1 but not for B_2. Lemma 2 implies that if r is not evaluated for B_2, then for B_2, $u[r] = u[s]$ for any s, even if evaluation of s loops. On the other hand, if evaluation of s loops, then $u[s] \neq u[r]$ for B_1, since evaluation of $u[r]$ terminates. Thus, B_1 and B_2 cannot be equivalent, and we have a contradiction. ∎

Lemma 3 (Substitutivity Lemma): Let P be a program and s be any ground term. If any subterm u of s is replaced by a ground subterm u' forming a new term s', and if $Evb[\![u]\!] = Evb[\![u']\!]$, then $Evb[\![s]\!] = Evb[\![s']\!]$.

Proof. We can restate the lemma as follows: Let $s = t[u]$ *for* x. If for some u', $Evb[\![u]\!] = Evb[\![u']\!]$, then $Evb[\![t[u]]\!] = Evb[\![t[u']]\!]$ *for* x.

The proof is by induction on the structure of t.

Base case. The term t is x. The result follows trivially.

Induction step. The term t is of the form $g(t_1, \ldots, t_n)$. Suppose x is a subterm of one of the arguments to g that occurs in a strict position. Let this term be t_i. By the induction hypothesis $Evb[\![t_i[u]]\!] = Evb[\![t_i[u']]\!]$ *for* x.

By the definition of leftmost-innermost and by Lemma 2, all the arguments $t_1, \ldots, t_{i-1}$ will be evaluated completely before any expansion takes place in $t_i[u]$ or $t_i[u']$. If this part of the evaluation process does not terminate, then trivially $Evb[\![t[u]]\!] = Evb[\![t[u']]\!]$. If it does terminate, then at some point the current term in $Evb[\![t[u]]\!]$ will have the form $g(c_1, \ldots, c_{i-1}, t_i[u], \ldots, t_n)$, and similarly the current term in $Evb[\![t[u']]\!]$ will eventually have the form $g(c_1, \ldots, c_{i-1}, t_i[u'], \ldots, t_n)$.

By Lemma 2, $t_i[u]$ (and $t_i[u']$) must be completely evaluated before any of the terms $t_{i+1}, \ldots, t_n$ are. If $Evb[\![t_i[u]]\!]$ (and thus $Evb[\![t_i[u']]\!]$) does not terminate, then none of the evaluations terminate, and again trivially $Evb[\![t[u]]\!] = Evb[\![t[u']]\!]$. If these evaluations do terminate, then both will have the same result, say c_i. Thus in both evaluations, the current term will eventually have the form $g(c_1, \ldots, c_i, t_{i+1}, \ldots, t_n)$, and so both of the results will be the same.

The case where x does not occur in a strict position of g is handled similarly.

Induction step. The term t is of the form $f(t_1, \ldots, t_n)$. Now x is a subterm of some argument to t, say t_i. The proof now follows that of the previous case. ∎

6. Evaluation Trees

We now define a notion of evaluation tree, which will provide an explicit representation of the history of an evaluation. These trees have a rather unusual structure, but they will help us reason about evaluation of programs with expression procedures. The three lemmas proved above enable us to define evaluation trees in a recursive manner.

Given a basic program P and a ground term t, an ***evaluation tree, Tree⟦t⟧***, is an ordered tree constructed as follows:

(a) If t is a constant symbol c, then the evaluation tree $Tree⟦t⟧$ consists of a single node labeled c.

(b) Suppose t is a term of the form $f(t_1, \ldots, t_n)$, where f has a definition of the form $f(x_1, \ldots, x_n) \leftarrow u$. Let $t_{i_1}, \ldots, t_{i_k}$, where $i_j < i_{j+1}$, be the arguments of f that correspond to those x_i that do *not* occur as strict subterms of the body u. Then the evaluation tree $Tree⟦t⟧$ consists of a root node labeled $f(t_1, \ldots, t_n)$, which has $k+1$ sons. The first son is the root of an evaluation tree for $u[t_1, \ldots, t_n]$ *for* $x_1, \ldots, x_n$. The other k sons are evaluation trees for the terms $t_{i_1}, \ldots, t_{i_k}$.

(c) If t is a term of the form $g(t_1, \ldots, t_n)$, then $Tree⟦t⟧$ has the following form: Let $t_{i_1}, \ldots, t_{i_k}$ where $i_j < i_{j+1}$ be the arguments to g that must be evaluated before $g(t_1, \ldots, t_n)$ simplifies to a constant. (These arguments will be $t_1, \ldots, t_j$ and possibly an additional term t_ℓ.) The evaluation tree consists of a node n labeled $g(t_1, \ldots, t_n)$, which has k sons, which are the evaluation trees for $t_{i_1}, \ldots, t_{i_k}$.

(Note: The k extra sons are added to nodes for derivative symbols to reflect the fact that in call-by-value evaluation, all the arguments to a function are evaluated before its body is.)

Observe that for a given basic program, there is exactly one evaluation tree for each term t. We will prove below that this tree is finite if and only if $Evb⟦t⟧$ terminates.

Let t be a ground term. The *value* of t, $value⟦t⟧$, is a term defined inductively on the structure of the evaluation tree for t.

(a) If the tree is a single node, the node is labeled with a primitive term t, and $value⟦t⟧$ is equal to $simp⟦t⟧$.

(b) If the root node m of the tree has a label of the form $f(t_1, \ldots, t_n)$, where there is a definition of the form $f(x_1, \ldots, x_n) \leftarrow u$, then the value of the tree is the value of the subtree rooted at the first son of m.

(c) Otherwise the root node m of the tree has a label that is of the form $g(t_1, \ldots, t_n)$. Let $c_{i_1}, \ldots, c_{i_k}$ be the values of the evaluation trees rooted at

the sons of m. The value of the tree is then

$$simp⟦g(x_1, \ldots, x_n)[c_{i_1}, \ldots, c_{i_k}] \text{ for } x_{i_1}, \ldots, x_{i_k}⟧.$$

An evaluation tree T_1 is a *subtree* of an evaluation tree T_2 if T_2 can be obtained from T_1 in the following manner:

(a) a leaf node labeled with a constant of an arbitrary tree T' is replaced by the root of the tree T_1, and

(b) any leaf nodes of T_1 labeled with constants are replaced by arbitrary evaluation trees.

Lemma 4: Basic facts about evaluation trees: Let P be any basic program.

(a) If t is a ground term that has a finite evaluation tree T, then evaluation trees for all the strict subterms of t occur as subtrees of T.

(b) If t is a ground term with a finite evaluation tree, then $Evb_P⟦t⟧$ terminates with result equal to the value of the tree.

(c) If evaluation of a ground term t terminates, then the evaluation tree for t is finite and $Evb⟦t⟧$ is equal to $value⟦t⟧$.

Proof of (a). We prove this for all ground terms by induction on the structure of trees.

Base case. The tree is a single node. Therefore, the term t is a term that simplifies directly to a constant. The result follows directly from the definition of strict subterm.

Induction step. The tree has a root node m labeled with a term t of the form $f(u_1, \ldots, u_n)$, where there is a basic definition $f(x_1, \ldots, x_n) \leftarrow u$. The strict subterms of t are the terms u_i and their strict subterms. By the construction, all the u_i occur as subtrees of the node for t or as subtrees of $u[u_1, \ldots, u_n]$. By the induction hypothesis, they have subtrees for their strict subterms.

Induction step. The tree has a root node labeled with a term t of the form $g(u_1, \ldots, u_n)$. The proof follows that of the previous case.

Proof of (b). The proof is by induction on the structure of the evaluation tree.

Base case. The tree is a single node. In this case, the term t simplifies directly to a constant, and the result follows trivially.

Induction step. The tree has root node m labeled with a term of the form $f(\bar{u})$, where there is a definition of the form $f(\bar{x}) \leftarrow u$. By Part (a) of this lemma, each of the terms u_i occurs as a subtree of the tree for $f(\bar{u})$, and so by the induction hypothesis, evaluations of these terms terminate with the correct results.

Therefore, by Lemma 3, evaluation of $f(\bar{u})$ is equivalent to that of $f(\bar{c})$ where each c_i is the result of evaluating a term u_i. Now, if the current term in evaluation is $f(\bar{c})$, then the next term will be $u[\bar{c}]$ *for* $\bar{x}$, which, again by Lemma 3, is equivalent to evaluation of $u[\bar{u}]$. But this is the label of the first son of m, and so evaluates to the correct result by virtue of the induction hypothesis.

Induction Step. The tree has root node m labeled with a term of the form $g(\bar{u})$. To simplify this term to a constant, some of the arguments may have to be evaluated first. By the definition of evaluation tree, there are sons of m for each of these arguments. Further, by the induction hypothesis, evaluation of each of these terminates with the correct result. Thus, by Lemma 3, $g(\bar{u})$ is equivalent to a term of the form $g(u'_1, \ldots, u'_n)$. Here each u'_i is either an appropriate c_i, if there is a son for u_i, or the term u_i itself, if not. By the definition of evaluation tree, this term simplifies directly to a constant.

Proof of (c). (Outline) We do a complete induction on the length of evaluation.

Induction step. We consider two cases, depending on the first term s in the evaluation sequence.

Case. The term s is a primitive term. In this case the tree is a single node with a label s. It follows from the definition of evaluation tree that $value[\![s]\!]$ is equal to the result of evaluation.

Case. The term s is of the form $t'[f(\bar{u})]$ *for* x where $f(\bar{u})$ is a leftmost-innermost subterm, and there is a definition $f(\bar{x}) \leftarrow u$. By Lemma 2 the current term will eventually be $t'[c]$ where c is the result of evaluating $f(\bar{u})$. Since the evaluation sequence for $t'[c]$ is shorter than that for s, we know

that by the induction hypothesis, its corresponding evaluation tree is finite and has the correct value.

Similarly, since evaluation of $f(\bar{u})$ terminates, so must evaluations for the u_i, by part (a), above, and evaluation of the substituted body, $u[\bar{u}]$. By the induction hypothesis, there are finite evaluation trees for each of these terms, and with the correct values.

We can thus construct the tree for s by modifying the tree for $t'[c]$. The nodes for c corresponding to the instance of x in t' are replaced by copies of the subtree for $f(u_1, \ldots, u_n)$, and the labels in the tree are appropriately modified. It can be proved by induction on the structure of the tree for $t'[c]$ that the tree is well-formed and has the correct value. ∎

An evaluation tree for $fib(2)$, where fib is defined as in the previous example, appears in the figure. (The subtree values are shown in brackets.)

7. Program Substitution

*The following theorem establishes that if terms are substituted for com*putationally equivalent terms in the definitions of a program, the new program that results may terminate for fewer inputs, but when it does terminate, the programs are equivalent. Thus, after a sequence of such replacement steps, we know that when the resulting program terminates, it will have the correct result.

This provides an operational proof of a similar statement in [Burstall77]. We will use this theorem when reasoning about the Burstall and Darlington, and the Manna and Waldinger, transformation systems.

Let P be a basic program. Two terms r and s are ***evaluation equivalent*** with respect to P if

$$Eval_P[\![r[\bar{u}]]\!] = Eval_P[\![s[\bar{u}]]\!] \qquad \text{for } \bar{x}$$

for any set of ground terms $\bar{u}$, where $\bar{x} = vars[\![r]\!] \cup vars[\![s]\!]$.

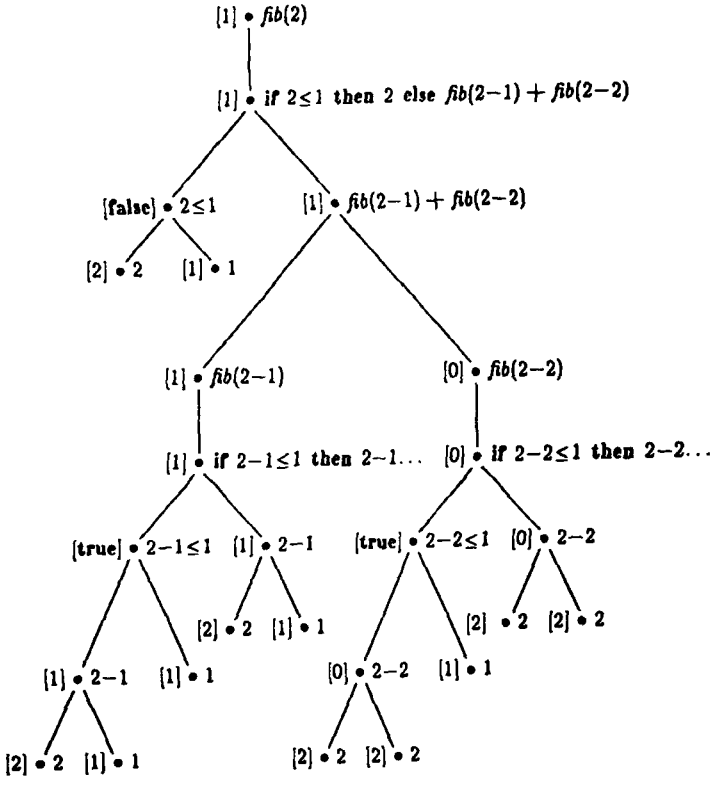


Figure: Evaluation tree for *fib*(2)

Theorem: Let P be a program containing a definition $f(\overline{x}) \leftarrow u[r]$. Let P' be a program identical to P except that the definition of f is replaced by a new well-formed definition $f(\overline{x}) \leftarrow u[s]$. If r and s are evaluation equivalent in P then, for any term t, if $Evb_{P'}[\![t]\!]$ terminates, then so does $Evb_P[\![t]\!]$, and with the same result.

Proof. Let t be a term such that $Evb_{P'}[\![t]\!]$ terminates. We use induction on the corresponding evaluation tree (by Lemma 4) to show that $Evb_P[\![t]\!]$ terminates, and with the same result.

Base case. The tree for t is a single node. In this case, t simplifies directly to a constant, and the result follows trivially.

Induction step. The root of the tree is a node m labeled with a term of the form $h(\overline{u})$, where there is a definition $h(\overline{x}) \leftarrow u$. We consider two cases:

Case. The derivative symbol h is not f. In this case, both P and P' share the same definition for h. The first son of m is labeled $u[\overline{u}]$, and evaluates equivalently in P and P' (by the induction hypothesis). The induction hypothesis also establishes that the "extra" sons have finite evaluations in P. Thus, the evaluation tree for $f(\overline{u})$ in P is finite and has the correct value. By Lemma 4, evaluation in P terminates with correct result.

Case. The derivitive symbol h is f. Now in P', the value of $f(\overline{u})$ is the value of the first son of m, which is $u[s][\overline{u}]$ *for* $\overline{x}$. That is,

$$Evb_{P'}[\![f(\overline{u})]\!] = Evb_{P'}[\![u[s][\overline{u}]]\!].$$

By the induction hypothesis,

$$Evb_{P'}[\![u[s][\overline{u}]]\!] = Evb_P[\![u[s][\overline{u}]]\!],$$

and by our assumptions and Lemma 3,

$$Evb_P[\![u[s][\overline{u}]]\!] = Evb_P[\![u[r][\overline{u}]]\!].$$

Since in P there is a definition $f(\overline{x}) \leftarrow u[r]$, we have

$$Evb_P[\![u[r][\overline{u}]]\!] = Evb_P[\![f(\overline{u})]\!],$$

by reasoning similar to that of the previous case. Thus, transitivity yields

$$Evb_{P'}[\![f(\overline{u})]\!] = Evb_P[\![f(\overline{u})]\!],$$

which is the desired result.

Induction step. The root of the tree is a node m labeled with a term of the form $g(\bar{u})$. By the induction hypothesis, evaluations of the appropriate terms u_i in P' terminate and with the correct results. Since the simplification sets for the two programs are the same, it follows that $g(\bar{u})$ evaluates to the same result for both programs. ∎

8. Well-founded Orderings

In order to prove that the transformation rules do not introduce loops into programs, it is necessary to introduce of a notion of computational progress. We do this by developing a measure on terms, mapping terms to the elements of a well-founded set.

A relation $\succ_w$ over a set W is a *well-founded ordering* if it is transitive and if there are no infinite sequences $w_1, w_2, w_3, \ldots$, of elements of W such that

$$w_1 \succ w_2 \succ w_3 \succ \ldots \quad .$$

The pair $(W, \succ_w)$ is called a *well-founded set.*

An example of a well-founded set is the nonnegative integers with the usual ordering, $>$. The proper subset relation on finite sets or multisets is also well-founded.

Well-founded orderings can also be constructed from other well-founded orderings. for example, Let $(W_1, \succ_1)$ and $(W_2, \succ_2)$ be two well-founded orderings. Then the *lexicographic ordering* over $W_1 \times W_2$ defined by

$$(w_1, w_2) \succ_{lex} (w_1', w_2') \quad \text{if} \quad w_1 \succ_1 w_1' \text{ or } (w_1 = w_1' \text{ and } w_2 \succ_2 w_2').$$

is well-founded.

It is well known that well-founded orderings can be used to prove termination of basic programs [Manna74]. We state this *termination principle* as follows:

A term of the form $f(\bar{c})$ is called a *ground basic term.*

Termination Principle: For any basic program B, there is a well-founded set $(W, \succ_w)$ and a mapping (the *termination mapping*) $K_0[\![t]\!]$ mapping ground basic terms to elements of W, such that Evb terminates for a ground term t if and only if the following condition is met: In the evaluation tree for t, if a node labeled $f_1(\bar{u}_1)$ is an ancestor of a node labeled $f_2(\bar{u}_2)$, then there are sets of constants $\bar{c}_1$ and $\bar{c}_2$ equal to the values of $\bar{u}_1$ and $\bar{u}_2$ respectively such that $K_0[\![f_1(\bar{c}_1)]\!] \succ K_0[\![f_2(\bar{c}_2)]\!]$.

Our restriction of K_0 to ground basic terms ensures that K_0 corresponds to the conventional notion of termination function, mapping argument *values* to the elements of a well-founded set.

Such a termination mapping always exists, though it may be difficult to construct. We can always define a termination function K_0 mapping ground basic terms to nonnegative integers in the following nonconstructive way: Let t be a any ground basic term. If evaluation of t terminates, then $K_0[\![t]\!]$ is the depth of the evaluation tree for t. If evaluation of t does not terminate, then $K_0[\![t]\!]$ is zero.

As a more practical example, consider the following factorial program, in which the domain of the variables is the set of integers:

$$fact(x) \leftarrow \text{if } x = 0 \text{ then } 1 \text{ else } x \cdot fact(x - 1)$$

Evaluation of $fact(x)$ terminates if and only if $x \geq 0$. One possible termination ordering for $fact$ maps terms of the form $fact(n)$ for an integer n to the set of natural numbers,

$$K_{fact}[\![fact(n)]\!] = \begin{cases} n, & \text{if } n > 0 \\ 0, & \text{otherwise.} \end{cases}$$

Note that the termination principle also applies to evaluations of "nested" terms. Given the well-founded ordering and termination mapping above, if we want to prove that evaluation of the term $fact(fact(2))$ terminates, we must first show that there exists a constant c whose value is equal to that of the argument of the outer call to $fact$; that is, we must show that evaluation of $fact(2)$ terminates. We can do this by applying the termination principle to this inner term.

In order to characterize the concept of computational progress, we show how the notion of termination mapping, whose domain is the set of ground basic terms, can be extended to a notion of *measure mapping* over all ground terms. We will thus be able to assign well-founded set values to complicated terms like $fib(2) + fib(1)$ (where fib is the Fibonacci function defined above).

It is apparent that such a mapping K must have at least the following two properties: First, the restriction of K to ground basic terms should be a termination ordering. Second, it is natural for K to have a "substitution" property: If there is a subtree for s in the evaluation tree for $t[s]$, then

$$K[\![s]\!] \gg K[\![u]\!] \qquad \text{implies} \qquad K[\![t[s]]\!] \gg K[\![t[u]]\!].$$

Thus, if $K[\![fib(3)]\!] \succ K[\![fib(2) + fib(1)]\!]$, then this property would imply that $K[\![fib(2) + fib(3)]\!] \succ K[\![fib(2) + (fib(2) + fib(1))]\!]$. These properties, which will be stated in precise fashion below, are called the *measure properties*.

To construct measure orderings we use a derived ordering, the multiset ordering of Dershowitz and Manna [Dershowitz79a].

Given a well-founded set $(W, \succ_W)$, the *sequential multiset ordering* $\gg_{M(W)}$ is the transitive closure of the relation $\succ$ defined as follows: If M and M' are multisets in $M(W)$, where $M(W)$ is the set of multisets over W, then $M \succ M'$ if, for some $x \in M$ and some multiset $Y \in M(W)$,

$$M' = (M - \{x\}) \uplus Y \qquad \text{and} \qquad (\forall y \in Y) \quad x \succ_W y.$$

That is, a multiset decreases when any element is replaced by zero or more smaller elements. (This is equivalent to the definition appearing in [Dershowitz79a].) Observe that if M_1 and M_2 are multisets such that $M_1 \supset M_2$, then for any underlying ordering W, it is true that $M_1 \gg_{M(W)} M_2$. We may occasionally use the notation $\uplus^i A$ to indicate the multiset union, $A \uplus \cdots \uplus A$, of i instances of the multiset A.

For example, let N be the set of nonnegative integers ordered by $>$. The following all hold in the multiset ordering $M(N)$:

$$\{1,1,1\} \gg \{1,1\} \gg \{1\} \gg \emptyset$$
$$\{3,2\} \gg \{3,1,1\} \gg \{2,2,1,1,1\} \gg \{1,1,1,1,1,1\}.$$

Lemma 5: Basic facts about the multiset ordering: Suppose $(W, \succ_W)$ is well-founded. Then we have the following results.

(a) $(M(W), \gg_{M(W)})$ is well-founded.

(b) For any sets $A, B, C, D \in M(W)$,

$$\begin{array}{rcl} A \gg B & \Leftrightarrow & A \uplus C \gg B \uplus C \\ A \gg B \ \wedge \ B \gg C & \supset & A \gg C \\ A \gg B \uplus C \ \wedge \ C \gg D & \supset & A \gg B \uplus D \\ A \gg B \ \wedge \ B \uplus C \gg D & \supset & A \uplus C \gg D \end{array}$$

(c) For any elements $a, b_1, \ldots, b_n \in W$,

$$a \succ b_1 \wedge \cdots \wedge a \succ b_n \quad \Leftrightarrow \quad \{a\} \gg \{b_1, \ldots, b_n\}.$$

Proof. Part (a) is proved in [D&M]. The proofs of (b) and (c) follow directly from the definition. Observe that the second two properties in (b) follow from the first two. ∎

We are now ready to precisely define the notion of measure function, using this multiset model.

A function K mapping ground terms to multisets over a well-founded set W is a *measure function* (or *measure ordering*) if it has the following two properties:

(a) (*Termination*) For any ground basic term $f(\bar{c})$ whose evaluation terminates and for which there is a definition $f(\bar{x}) \leftarrow u$,

$$K[\![f(\bar{c})]\!] \gg K[\![u[\bar{c}] \text{ for } \bar{x}]\!],$$

(b) (*Substitution*) For any ground term $t[u]$ whose evaluation terminates,

$$K[\![t[u]]\!] = K[\![t[c]]\!] \ \uplus \ (\uplus^i K[\![u]\!]),$$

where i is a function of $t[c]$ and greater than zero if and only if there is a subtree for u in an evaluation tree for $t[u]$. When $i > 0$, the constant c must be equivalent to the value of the tree for u.

These two properties are called the *measure properties*.

Observe that these properties imply that the restriction to ground basic terms of any measure mapping is a termination ordering. That is, any measure

ordering is a termination ordering. Observe also that for any terms t_1 and t_2, if the evaluation tree for t_2 is a subtree of the evaluation tree for t_1 then $K[\![t_2]\!] \ll K[\![t_1]\!]$, where K is any measure mapping.

Note also that the substitution property implies that for any ground terms $t[s]$ and $t[u]$, if there is a subtree for s in the evaluation tree for $t[s]$, then

$$K[\![s]\!] \gg K[\![u]\!] \quad \text{implies} \quad K[\![t[s]]\!] \gg K[\![t[u]]\!];$$

otherwise $K[\![t[s]]\!] = K[\![t[u]]\!]$.

It is straightforward to construct measure functions for basic programs. Let P be a basic program, and let M be any mapping from ground basic terms to nonempty multisets over a well-founded set W. The mapping K_P from ground terms to multisets over W is defined inductively over the structure of (finite) evaluation trees:

(a) If the term t is a constant, then $K_P[\![t]\!] = \emptyset$, the empty multiset.

(b) If the term t is of the form $f(\bar{u})$, where there is a definition $f(\bar{x}) \leftarrow u$, then

$$K_P[\![f(\bar{u})]\!] = M[\![f(\bar{c})]\!] \uplus K_P[\![u[\bar{u}] \text{ for } \bar{x}]\!] \uplus K_P[\![u_{i_1}]\!] \uplus \cdots \uplus K_P[\![u_{i_k}]\!],$$

where $\bar{c}$ are the values of the evaluation subtrees for the $\bar{u}$, and the terms u_{i_j} correspond to those u_i that occur in nonstrict positions in $u[\bar{u}]$.

(c) If the term t is of the form $g(\bar{u})$, then

$$K_P[\![g(\bar{u})]\!] = K_P[\![u_{i_1}]\!] \uplus \cdots \uplus K_P[\![u_{i_k}]\!],$$

where u_{i_j} are the labels of the sons of the root of the evaluation tree for $g(\bar{u})$.

As an example, we consider again the Fibonacci program. We can define a measure mapping K in terms of a function M that maps all ground terms to the singleton multiset $\{1\}$. Thus,

$$\begin{aligned} K[\![fib(3)]\!] &= \{1\} \uplus K[\![\textbf{if } 3 \le 1 \textbf{ then } 3 \textbf{ else } fib(3-1)+fib(3-2)]\!] \\ &= \{1\} \uplus K[\![fib(2)]\!] \uplus K[\![fib(1)]\!] \\ &= \{1\} \uplus \{1\} \uplus K[\![\textbf{if } \ldots]\!] \uplus \{1\} \uplus K[\![\textbf{if } \ldots]\!] \\ &= \{1\} \uplus \{1\} \uplus K[\![fib(1)]\!] \uplus K[\![fib(0)]\!] \uplus \{1\} \\ &= \{1\} \uplus \{1\} \uplus \{1\} \uplus \{1\} \uplus \{1\} \\ &= \{1,1,1,1,1\} \end{aligned}$$

The basic idea of the proofs of the transformation rules will be this: We have shown that every basic program has a measure function K. We will prove that if a program has a measure function before application of a transformation rule, then it will have one afterward. We will use this to prove that any sequence of transformation steps starting from a basic program preserves equivalence of programs.

9. Progressive Pairs

Given a program P and a measure ordering K, a ***ground progressive pair*** is a pair $\langle t_1, t_2 \rangle$ of ground terms such that,

(a) (*consistency*) $Evb[\![t_1]\!] = Evb[\![t_2]\!]$, and

(b) (*progressiveness*) if evaluation of t_1 and t_2 terminates then $K[\![t_1]\!] \gg K[\![t_2]\!]$.

A *progressive pair* is a pair $\langle t_1, t_2 \rangle$ of terms such that any ground instance

$$\langle t_1[\bar{u}], t_2[\bar{u}] \rangle \text{ for } \bar{x}$$

of the pair is a ground progressive pair.

Observe that any instance of a progressive pair is also a progressive pair. We will eventually prove that all expression procedures created by the transformation rules are progresssive pairs.

For a given program with a definition $s \leftarrow t$, the pair

$$\langle s[\bar{u}],\ t[\bar{u}] \rangle \text{ for } \bar{x}$$

where $x_1, \ldots, x_n \in vars[\![s]\!]$, is a *proper definition instance* of $s \leftarrow t$ if for each term u_i, either x_i occurs as a strict subterm of t, or evaluation of u_i always terminates.

As an example, any instance of $\langle fib(x), fib(x-1)+fib(x-2) \rangle$, such that $x > 1$, is a ground progressive pair. The pair,

$$\langle fib(fib(x)),\quad \textbf{if } fib(x) \le 1 \textbf{ then } fib(x) \textbf{ else } fib(fib(x)-1)+fib(fib(x)-2) \rangle,$$

is a proper definition instance of *fib*, if x is a variable over the nonnegative integers. Note however that the definition $f(x) \leftarrow 3$ is not a proper definition instance of itself.

Lemma 6: For a given basic program and measure ordering, a proper definition instance (r, s) of a definition $f(\bar{x}) \leftarrow t$ forms a progressive pair.

Proof. A ground instance of a proper definition instance of $f(\bar{x}) \leftarrow t$ has the form

$$(f(\bar{u}),\ t\,[\bar{u}]\ \textit{for}\ \bar{x})$$

where for each i, u_i is ground, and either x_i occurs as a strict subterm of t or evaluation of u_i terminates.

An evaluation tree for $f(u_1, \ldots, u_n)$ consists of a root m that has as sons evaluation trees for $t[\bar{u}]$ and $u_{i_1}, \ldots, u_{i_k}$, where the variables x_{i_j} do not occur as strict subterms in t. By the definition of proper definition instance, evaluation of all instances of each term u_{i_j} terminates, and so these subtrees are finite. Thus, the tree for $f(\bar{u})$ is finite if and only if the one for $t[\bar{u}]$ is.

Since the value of the evaluation tree for $f(\bar{u})$ is $t[\bar{u}]$, we have the consistency condition: $Evb[\![f(\bar{u})]\!] = Evb[\![t[\bar{u}]]\!]$.

The progressiveness condition follows directly from the measure properties since $Tree[\![t[\bar{u}]]\!]$ is a subtree of $Tree[\![f(\bar{u})]\!]$. ∎

10. Evaluation for Full Programs

We now extend the evaluator for basic programs to one for full programs, with expression procedures. Given a program P and a set of primitive simplification rules S, the procedure $Evc_P[\![t]\!]$ on legal input terms for P is defined as follows:

Evc[[t]]:

(1) (*Simplify*) Let t be $simp[\![t]\!]$.

(2) If t is now primitive, then exit the evaluator with t as the result. (Otherwise, t contains a derivative name.)

(3) (*Choose*) Since P is well-formed, there is at least one definition $s \leftarrow u$ that has a proper definition instance (s', u') such that the leftmost-innermost choice $f(t_1, \ldots, t_n)$ occurs as a strict subterm of s' in

$$t = t'[s']\ \textit{for}\ x$$

where x is leftmost in t'. (The basic definition for f trivially has this property.)

(4) (*Expand*) Let t be $t'[u']$ *for* x.

(5) Go to step (1). ∎

If this evaluation procedure terminates, its result will be a constant symbol.

Unlike Evb, this evaluator is nondeterministic. A nondeterministic evaluator has the *universal termination property* for a program if, for every legal input term t, either all computation paths terminate, and with the same result, or none of them do.

Two programs P_1 and P_2 are *equivalent* if they both have the universal termination property and their basic segments are equivalent.

A program P is *uniform* if there is a well-founded ordering K such that (a) K is a measure ordering for P, and (b) all the complex definitions of P are progressive pairs in K.

Observe that any basic program is uniform.

Consider, for example, the basic program B: $f(x) \leftarrow x$; $h(x) \leftarrow x$. If the complex definition, $f(g(z)) \leftarrow h(g(z))$ is added to B then the resulting program P_1 is uniform. Similarly, if the complex definition $h(g(z)) \leftarrow f(g(z))$ is added to B, then the resulting program P_2 is uniform. If, however, both definitions are added to B, then the resulting program is *not* uniform.

As another example, consider the uniform program

$$fib(x) \leftarrow \textbf{if } x \leq 1 \textbf{ then } x \textbf{ else } fib(x-1) + fib(x-2)$$
$$fib(x+1) + fib(x) \leftarrow \textbf{if } x \leq 1 \textbf{ then } x+1 \textbf{ else } 2 \cdot fib(x) + fib(x-1)$$

Two possible evaluation paths for $fib(3)$ are

$fib(3)$	$fib(3)$
$fib(2) + fib(1)$	$fib(2) + fib(1)$
$2 \cdot fib(1) + fib(0)$	$fib(1) + fib(0) + fib(1)$
$2 + fib(0)$	$1 + fib(0) + fib(1)$
2	$1 + fib(1)$
	2

Lemma 7. All proper definition instances of definitions in a uniform program are progressive pairs.

Proof. If the original definition is a basic definition, then this follows directly from Lemma 6. Otherwise it is a complex definition and thus a progressive pair itself. The result then follows directly from the definition of progressive pair. ∎

Lemma 8. *Evc* has the universal termination property for uniform programs.

Proof. It follows from the definition of *Evc* that for a given ground term t, there is a computation path for *Evc* that corresponds to the computation path of the basic evaluator, *Evb*. We call this path the ***basic path***. It suffices to show that the results of all computation paths are equivalent to that of the basic path

Suppose first that the basic path does not terminate. It follows from the consistency property of the complex definitions of P, and from Lemma 3 that this termination property is not disturbed by the application of a complex definition (Recall that, since P is uniform, all of its complex definitions are progressive pairs.) Therefore, no other path terminates.

This leaves the case where the basic path is finite Again by consistency and Lemma 3 we know that application of a complex definition does not disturb the result of a path We must prove, however, that only finitely many complex definitions can be applied on any one path, thus ensuring termination of that path

We prove that each evaluation path is finite by exhibiting a mapping of current terms to the elements of a well-founded set such that each step

of evaluation is reflected by a corresponding decrease in the well-founded ordering. The mapping we use is the measure ordering K, mapping current terms to multisets. We know such an ordering exists since P is uniform. Let t be a current term in *Evc*. For every possible term t' that can arise from t after a step of evaluation, we must show that

$$K[\![t]\!] \gg K[\![t']\!].$$

A step of evaluation has the effect of replacing a current term $t = r[s']$ *for* x by a new term $t' = r[u']$ *for* x, where x occurs exactly once in r and (s', u') is a (ground) proper definition instance of a definition $s \leftarrow u$. (In fact, each u_i in the definition is a primitive term.)

Since (s', u') is a ground progressive pair (by Lemma 7), we have that $K[\![s']\!] \gg K[\![u']\!]$; and so by the substitution property of K, we have

$$K[\![r[s']]\!] \gg K[\![r[u']]\!].$$

∎

11. The Transformation Rules

The four transformation rules can now be stated and proved. We make use of the concepts of strict subterm and proper definition instance in the statement of the rules (see sections 4 and 9, respectively). All initial programs are assumed to be uniform.

Composition. Let $s \leftarrow u$ be a definition in a given program P, and (s', u') a proper definition instance. *Composition* produces a new program P', that is the program P with an additional complex definition

$$t[s'] \leftarrow t[u'] \quad \textit{for } z;$$

here the variable z occurs exactly once in the given term t, and it is a strict subterm of t.

Notes The pair (s', u') must be a proper definition instance to prevent composing a definition like $f(x) \leftarrow 3$ into $f(k(x)) \leftarrow 3$ in the presence of $k(x) \leftarrow k(x)$. The pair $\langle f(k(x)), 3 \rangle$ is not progressive since evaluation of $f(k(x))$ never terminates while evaluation of 3 always does. In addition, z must be a strict subterm to prevent composing a definition like $k(x) \leftarrow k(x)$ into

$$\textbf{if } x < 1 \textbf{ then } h(x) \textbf{ else } k(x) \leftarrow \textbf{if } x < 1 \textbf{ then } h(x) \textbf{ else } k(x),$$

thus introducing a loop where there might have been none before.

Application. Let $s \leftarrow u$ be a definition in a given program P, and $\langle s', u' \rangle$ a proper definition instance. *Application* produces a new program P' that is the program P with a definition,

$$t \leftarrow t'[s'] \text{ for } z,$$

replaced by the new definition

$$t \leftarrow t'[u'] \text{ for } z,$$

where the variable z occurs exactly once in the term t'.

Notes The pair $\langle s', u' \rangle$ must be a proper definition instance to prevent application from transforming the definition $h(x) \leftarrow f(k(x))$ into $h(x) \leftarrow 3$ in the presence of $f(x) \leftarrow 3$ and $k(x) \leftarrow k(x)$. This will cause evaluation of h to terminate where it did not previously.

Abstraction. Let

$$s_1 \leftarrow t_1[t'[u_1]] \text{ for } \bar{x}] \quad \text{for } z$$

$$\vdots$$

$$s_m \leftarrow t_m[t'[\bar{u}_m] \text{ for } \bar{x}] \quad \text{for } z$$

be definitions in a given program P. *Abstraction* produces a new program with these definitions replaced by the new definitions,

$$s_1 \leftarrow t_1[f(\bar{u}_1)] \quad \text{for } z$$

$$\vdots$$

$$s_m \leftarrow t_m[f(\bar{u}_m)] \quad \text{for } z$$
$$f(\bar{x}) \leftarrow t',$$

where

(a) f is a derivative name that does not occur anywhere in P;

(b) $vars[\![t']\!] = \{x_1, \ldots, x_n\}$;

(c) Each x_i occurs as a strict subterm of t'.

Notes: The second restriction ensures well-formedness of the new definition. The restriction on x_i is designed to prevent abstractions of the form,

$$P(x) \leftarrow \textbf{if } Q(x) \textbf{ then } h(x) \textbf{ else } k(x)$$

into $P(x) \leftarrow f(x, k(x))$ and $f(u, v) \leftarrow$ **if** $P(u)$ **then** $h(u)$ **else** v; if evaluation of $k(x)$ loops, then this abstraction may cause a nonterminating evaluation path where there was none previously.

Definition Elimination. (a) Any expression procedure can be dropped from a given program. (b) Any basic definition that defines a derivative name that is not principal and that does not occur in the body part of any other definition can be dropped from a given program.

It is clear that all of these rules preserve well-formedness of programs. We prove below that equivalence is preserved.

12. Correctness of the Transformation Rules

Main Theorem: The four transformation rules (composition, application, abstraction, and definition elimination) preserve equivalence and uniformity of uniform programs.

Proof. To prove a rule correct, we must establish two facts: First, that the basic segment of the new program is equivalent to that of the old. Second, that the complex definitions are all still progressive pairs with respect to some measure function. This will imply that the old and new programs are equivalent and that the new program is uniform.

Composition. The composition rule adds a single complex definition, but does not change any basic definitions. Thus, the new program has a measure

function K that is the same as that of the initial program. We must show only that the new definition $t[s'] \leftarrow t[u']$ is a progressive pair, given that the initial definition $s \leftarrow u$ is one. This will establish uniformity and thus equivalence of the new program.

The consistency of the new pair follows from Lemma 3.

We prove progressiveness as follows: Let (s'', u'') be any ground instance of the proper definition instance (s', u') such that evaluation of s'' terminates. Since s' is a strict subterm of $t[s']$, s'' must be evaluated when $t[s'']$ is. Further, the initial definition $s \leftarrow u$ is a progressive pair, so $K[\![s'']\!] \gg K[\![u'']\!]$. Thus, by the substitution property of K we have $K[\![t[s'']]\!] \gg K[\![t[u'']]\!]$.

Application. The application rule modifies a single definition. We consider two cases.

Case If the definition is a complex definition, then the basic segment of the program does not change. Thus, the measure ordering K for the initial program is one for the final program as well, and it suffices to show that the definition affected by the rule is still a progresssive pair.

The application rule can then be considered to have two steps: First, a single complex definition is removed. This maintains equivalence (as proved below). Second, a new complex definition is added. We must show that this new definition forms a progressive pair.

Since the program is uniform, we know that the initial complex definition $t \leftarrow t'[s']$ forms a progressive pair, $(t, t'[s'])$. In addition, we know that (s', u') is a progressive pair. Therefore, by Lemma 3, the pair $(t, t'[u'])$ is consistent.

No strictness restrictions were placed on the term t'. Therefore, a subtree for s' may occur in evaluation trees for some ground instances of $t'[s']$ but not in others. Let $t''[s'']$ be an arbitrary ground instance. If a subtree does not occur, then $K[\![t''[s'']]\!] = K[\![t''[u'']]\!]$ by the substitution property. If it does occur, then clearly $K[\![t''[s'']]\!] \gg K[\![t''[u'']]\!]$, also by substitution. In either case, progressiveness then follows from the progressiveness condition of the original definition.

Case. If the definition modified is a primitive definition $f(\bar{x}) \leftarrow t'[s']$ *for* x, then it is necessary to prove equivalence of the basic segments of the initial program P and the resulting program P'. Let B and B' be the basic segments of P and P' respectively.

To prove equivalence of the B and B', we must prove co-termination and weak equivalence (see Section 5). Weak equivalence follows directly from the program substitution theorem. That theorem also implies that evaluations using B terminate whenever those of B' do. Thus, we must prove only the converse. But this follows if we show that the measure ordering K for B is also a measure ordering for B'. To do this we must show that the measure properties, termination and substitution, still hold.

Consider a definition $h(\bar{x}) \leftarrow u$ in B'. If h is not the same as f, then there is an identical definition in B for which the termination condition holds. Thus the termination condition holds for this definition as well. If h is the same as f, then the definition is of the form $f(\bar{x}) \leftarrow t'[u']$. But the termination condition holds for a corresponding definition $f(\bar{x}) \leftarrow t'[s']$ in B, and (s', u') is a progressive pair; thus the termination condition holds for this new definition. This implies that the termination condition on K holds for the new program P'.

It follows from the corollary to Lemma 2 and the substitution property of K for B that K has the substitution property for B'. Thus, K is a measure map for P', and so P' is uniform. ∎

Abstraction. The abstraction rule modifies one or more existing definitions and adds a new definition. The proof is similar to the one for application, except in this case the measure ordering K for P is modified into a new measure ordering K' that holds both for P and P'.

The basic segments B and B' (defined as above) are weakly equivalent by Lemma 3 and the program substitution theorem. We must prove that if basic evaluation in B terminates, then so does evaluation in B', and that the complex definitions of P' are consistent and progressive.

The ordering K is transformed into the ordering K' by modifying the underlying measure mapping M to account for the introduction of the new derivative name f. Recall that M maps ground basic terms into multisets of elements of some well-founded set $(W, \succ_W)$. We define a new well-founded set $(W', \succ_{W'})$ such that $W' = W \uplus \{\mathbf{f}\}$ and, for each $w \in W$, $w \succ_{W'} \mathbf{f}$.

Clearly W' is well-founded, as is K'. (This ordering is a *union* ordering, as defined in [Manna81].)

We now extend the definition of K to cover all ground terms involving f. By virtue of the substitution property of K, we need only explicitly extend the definition to include ground terms of the form $f(\bar{u})$. As in the definition of the basic measure mapping, K_P, we define

$$K'[\![f(\bar{u})]\!] = \{\mathbf{f}\} \uplus K'[\![t'[\bar{u}] \text{ for } \bar{x}]\!].$$

(There are no special u_{i_j} terms since all parameters to f are strict subterms of the terms in $\bar{u}$.)

Clearly, if t is a ground term not containing f, then $K'[\![t]\!] = K[\![t]\!]$. It thus follows that if t_1 and t_2 contain no instances of the new name f, then

$$K[\![t_1]\!] \gg K[\![t_2]\!] \quad \text{implies} \quad K'[\![t_1]\!] \gg K'[\![t_2]\!].$$

This implies that all unaffected basic definitions have the termination property, and all unaffected expression procedures are (consistent and) progressive. Thus, K' is a measure mapping for the initial program P.

To prove that K' is a measure mapping for the new program P', we must show that the termination and substitution properties hold for the definitions that have been altered by the abstraction operation.

By the corollary to Lemma 2, the substitution property is unaffected for all terms $t[u]$ in P' in which t contains no instances of f. In the case in which t contains an instance of $f(\bar{x})$, the substitution property follows (by an obvious induction on the structure of t) from the definition of K' for f and the substitution property of K.

It remains to show that K' has the termination property for P'. The definitions modified include the new basic definition and the definitions for $s_1, \ldots, s_m$. The termination property for the new basic definition, $f(\bar{x}) \leftarrow t'$, follows directly from the definition for $K'[\![f(\bar{u})]\!]$ above.

We now prove that the modified basic definitions have the termination property, and that the modified complex definitions are progressive. For a basic or complex definition $s \leftarrow t[f(\bar{u})]$, we must show that if evaluation of

$s[\bar{r}]$ terminates, then

$$K'[\![s[\bar{r}]]\!] \gg K'[\![t[f(\bar{u})][\bar{r}]]\!],$$

where $\bar{r}$ are all constants if s is basic. (If s is basic then this is the termination property; if s is complex, then this is the progressiveness property.) By the substitution property, this is equivalent to

$$K'[\![s[\bar{r}]]\!] \gg K'[\![t[c][\bar{r}]]\!] \uplus (\uplus^i K'[\![f(\bar{u})[\bar{r}]]\!]),$$

where c is the value of $f(\bar{u})[\bar{r}]$ if evaluation of f teminates. (Otherwise, this instance of f is not evaluated, and, by Lemma 2, the value of c is irrelevant.)

By the definition of K' for f, it follows that

$$K'[\![s[\bar{r}]]\!] \gg K'[\![t[c][\bar{r}]]\!] \uplus (\uplus^i K'[\![t'[\bar{u}][\bar{r}]]\!]) \uplus (\uplus^i\{\mathbf{f}\}),$$

Using the substitution property again, we obtain

$$K'[\![s[\bar{r}]]\!] \gg K'[\![t[t'[\bar{u}]][\bar{r}]]\!] \uplus (\uplus^i\{\mathbf{f}\}),$$

as the inequality to be proved.

Now, since $s \leftarrow t[t'[\bar{u}]]$ is a definition in P, we have

$$K'[\![s[\bar{r}]]\!] \gg K'[\![t[t'[\bar{u}]][\bar{r}]]\!].$$

But the definition of W' and the properties of the multiset ordering imply that for any multisets A and B over W, and for any i,

$$A \gg B \quad \text{implies} \quad A \gg B \uplus (\uplus^i\{\mathbf{f}\}),$$

so we are done.

We have thus proved that the termination property holds for the definitions of B', and that the modified complex definitions of P' are progressive. It follows that K' has the measure property for P' and that the basic programs B and B' are equivalent. Therefore the unmodified complex definitions are consistent. Consistency of the modified complex definitions is implied by Lemmas 3 and 6.

Thus, all the complex definitions of P' form progressive pairs; therefore we conclude that P' is uniform with respect to K'. ∎

Definition Elimination. The evaluator *Evc* is defined in such a way that removal of a definition from a program can never cause new evaluation paths to be added or existing ones to be altered; it can only cause existing paths to be removed. We therefore must show only that after application of the definition elimination rule, there are still computation paths for all legal input terms to the initial program P.

Since there is always an evaluation path for *Evc* in which only basic definitions are applied, any complex definition may be removed without ill effect. This establishes correctness of the first definition elimination rule.

If a derivative name is not principal, then the only way its definition can be applied in the evaluation of a legal input term, is if the defined name occurs in the body of an existing definition. Therefore, if the definition name does not occur in any other definition body, then its definition can be dropped. This proves correctness of the second definition elimination rule. ∎

13. Transformation Rules for Call-by-Name

The following pair of evaluators, for basic and full programs respectively, use the *call-by-name*, or *leftmost-outermost*, choice algorithm. While this choice algorithm does not correspond with many practical programming language implementations, it does appears frequently in theoretical presentations, and so it is instructive to state the transformation rules for this case.

Let B be a basic program with a set of primitive simplification rules S, and let t be a ground term.

$Evbn_B[\![t]\!]$:

(1) (*Simplify*) Let t be $simp_S[\![t]\!]$.

(2) If t is now primitive, then exit the evaluator with t as the result.

(3) (*Choose*) Since the program B is well-formed, there is a definition $f(\bar{x}) \leftarrow s$ such that there is a term t',

$$t = t'[f(\bar{u})] \text{ for } z,$$

where z occurs exactly once in t', and in a leftmost position such that it is on the left of any leftmost nonprimitive term in t'. (f is a *leftmost-outermost* choice.)

(4) (*Expand*) Let t be $t'[s[\bar{u}] \text{ for } \bar{x}] \text{ for } z$.

(5) Go to step (1). ∎

Evaluation trees for *Evbn* are defined similarly to those for *Evb*, except that for nodes labeled by $f(\bar{u})$, there are no special sons for Lonstrict u_{i_j}.

Let P be a (full) program, S as set of primitive simplification rules, and t a ground term.

$Evcn_P[\![t]\!]$:

(1) (*Simplify*) Let t be $simp_S[\![t]\!]$.

(2) If t is now primitive, then exit the evaluator with t as the result.

(3) (*Choose*) Since P is well-formed, there is at least one definition $s \leftarrow u$ that has a proper definition instance (s', u') such that the leftmost-outermost choice $f(\bar{t})$ occurs as a strict subterm of s' in

$$t = t'[s'] \text{ for } z$$

where z is leftmost in t'. (The basic definition for f trivially has this property.)

(4) (*Expand*) Let t be $t'[u'] \text{ for } z$.

(5) Go to step (1). ∎

The call-by-name transformation rules are generally less restrictive than the call-by-value rules (with the exception of composition). This follows from the fact that in call-by-name the arguments of a name are not evaluated in advance. Thus, we make use of a much more liberal definition of definition instance:

A *call-by-name definition instance* of a definition $s \leftarrow t$ is any pair

$$(s[\bar{u}], u[u]) \text{ for } \bar{x}$$

where $x_1, \ldots, x_n \in vars[\![s]\!]$.

A subterm s of a term $t[s]$ is *completely strict* if there are evaluation subtrees for s in evaluation trees for all instances of $t[s]$. (For call-by-value, this includes the notion of strict subterm. For call-by-name, however, the set of completely strict subterms of a term may be *smaller* than the set of strict subterms.)

The notions of progressive pair and uniformity are defined as in the call-by-value case. The transformation rules for call-by-name can now be stated. As before, initial programs are assumed to be uniform.

Composition. Let $s \leftarrow u$ be a definition in a given program P, and $\langle s', u' \rangle$ a call-by-name definition instance. *Composition* produces a new program P' that is the program P with an additional complex definition

$$t[s'] \leftarrow t[u'] \quad \textit{for } x,$$

where the variable x occurs exactly once in a given term t and is a completely strict subterm of t.

Notes: If x is not a completely strict subterm of t, then looping evaluation can be obtained where there were none previously. Suppose we have the two definitions, $k(z) \leftarrow k(z)$ and $f(z) \leftarrow 3$. Observe that $k(z)$ in not a completely strict subterm of $f(k(z))$. Without the restriction, we could compose to obtain the definition $f(k(z)) \leftarrow f(k(z))$, which enables the evaluator to loop where it did not previously.

Application. Let $s \leftarrow u$ be a definition in a given program P, and $\langle s', u' \rangle$ a call-by-name definition instance. *Application* produces a new program P' that is the program P with a definition,

$$t \leftarrow t'[s'] \textit{ for } x,$$

replaced by the new definition

$$t \leftarrow t'[u'] \textit{ for } x,$$

where the variable x occurs exactly once in the term t'.

Abstraction. Let

$$\begin{aligned} s_1 &\leftarrow t_1[t'[\overline{u}_1] \textit{ for } \overline{x}] \quad \textit{for } x \\ &\vdots \\ s_m &\leftarrow t_m[t'[\overline{u}_m] \textit{ for } \overline{x}] \quad \textit{for } x \end{aligned}$$

be definitions in a given program P. *Abstraction* produces a new program with these definitions replaced by the new definitions,

$$\begin{aligned} s_1 &\leftarrow t_1[f(\overline{u}_1)] \quad \textit{for } x \\ &\vdots \\ s_m &\leftarrow t_m[f(\overline{u}_m)] \quad \textit{for } x \\ f(\overline{x}) &\leftarrow t', \end{aligned}$$

where

(a) f is a derivative name that does not occur anywhere in P;

(b) $vars[\![t']\!] = \{x_1, \ldots, x_n\}$.

Definition Elimination. (a) Any expression procedure can be dropped from a given program. (b) Any basic definition that defines a derivative name that is not principal and that does not occur in the body part of any other definition can be dropped from a given program.

Chapter 3

Program Transformation Techniques

In this chapter, we discuss the application of the transformation rules to program derivation, with a special emphasis on the process of specialization. In the course of this investigation, we consider various extensions to both the language of programs and the fundamental set of transformation rules. Many examples are presented to illustrate the use of the rules.

One of the extensions considered is conditional or *qualified* definitions: Qualifiers allow definitions to be specialized to contexts that cannot be described strictly in terms of syntactic pattern matching. (A similar notion is used in [Wegbreit76].)

In the more difficult program derivation examples, it is often necessary to give explicit proofs of certain properties of the programs being manipulated. We will describe informally a notation for assertions in recursive programs, and an induction rule that makes use of it. Qualifiers and assertions thus provide a means of expressing pre-conditions and post-conditions on definitions.

1. Using the Composition Rule

We can think of the composition rule as a *context specification* rule, used to introduce expression procedures corresponding to special cases of other procedures. Some examples will illustrate this.

Consider the following recursive procedure for appending two lists,

$$s \circ t \leftarrow \textbf{if } s = \Lambda \textbf{ then } t \textbf{ else } cons(hd(s), tl(s) \circ t).$$

Here we assume that the list manipulating functions *hd*, *tl*, and *cons* are primitive, and that we have a set of primitive simplification transformations to simplify terms involving them. Observe that if s and t are lists, then evaluation of this function will always terminate. We can develop several specialized versions of this *append* function.

First suppose that we call this function in a particular context in which we know the list s to be the empty list, Λ. We can form a specialized version of *append* for this particular context by forming an expression procedure for the phrase $\Lambda \circ t$. Using the composition rule, we get

$$\Lambda \circ t \leftarrow \textbf{if } \Lambda = \Lambda \textbf{ then } t \textbf{ else } cons(hd(\Lambda), tl(\Lambda) \circ t).$$

Since $\Lambda = \Lambda$ is true, this simplifies directly to

$$\Lambda \circ t \leftarrow t.$$

(In nonrecursive cases such as this, we could get the same effect by applying the original expression procedure to each instance of the specialized call, and simplifying in context. This specialization approach has the advantage, however, that we do the simplification only once. In addition, we will see that in-line substitution does not work in the case where the specialized function is recursive.)

We can similarly write an expression procedure for the opposite case, in which the first argument to *append* is always nonempty. Since any nonempty list can be written as $cons(a, c)$, for some a and c, we can compose to get

$$cons(a, c) \circ t \leftarrow \textbf{if } cons(a, c) = \Lambda \textbf{ then } t \textbf{ else } cons\big(hd(cons(a, c)), tl(cons(a, c)) \circ t\big).$$

Since the result of *cons* is always nonempty,

$$(cons(a, b) = \Lambda) \quad = \quad \textbf{false}$$

(for all lists a and b), the conditional can be replaced by its else clause, giving

$$cons(a, c) \circ t \leftarrow cons\big(hd(cons(a, c)), tl(cons(a, c)) \circ t\big).$$

Now, using the properties of *hd*, *tl*, and *cons*,

$$hd(cons(a,c)) = a$$
$$tl(cons(a,c)) = c,$$

for any lists a and c, we obtain

$$cons(a,c) \circ t \leftarrow cons(a, c \circ t).$$

(The properties of primitive names used in these derivations are represented by primitive simplification transformations. Details are in Section 2.3.)

These two expression procedures can be used to produce a specialized version of *append* for the case in which the first argument is a singleton list. Since any singleton list can be written as $cons(a,\Lambda)$, for some a, we will start with our expression procedure for $cons(a,c) \circ t$ and specialize it to the case in which $c = \Lambda$. We compose to get

$$cons(a,\Lambda) \circ t \leftarrow cons(a, \Lambda \circ t).$$

Now, we can use the application rule to unfold the call to the expression procedure for $\Lambda \circ t$, giving

$$cons(a,\Lambda) \circ t \leftarrow cons(a,t).$$

This example illustrates how new expression procedures can be composed out of ones that have been previously introduced.

In these examples, we specialized procedures based on our knowledge of the forms of their arguments. We can also specialize based on knowledge of the use of the result. Suppose, for example, that in some context in which we are appending two lists together, we will only require the first element of the result. That is, for some s and t, we need compute only $hd(s \circ t)$. The composition rule produces the expression procedure,

$$hd(s \circ t) \leftarrow hd(\textbf{if } s = \Lambda \textbf{ then } t \textbf{ else } cons(hd(s), tl(s) \circ t)).$$

Bringing the call to *hd* inside the conditional, we get,

$$hd(s \circ t) \leftarrow \textbf{if } s = \Lambda \textbf{ then } hd(t) \textbf{ else } hd(cons(hd(s), tl(s) \circ t)).$$

Since s and t are assumed to be lists, evaluation of $tl(s) \circ t$ terminates, and this simplifies to

$$hd(s \circ t) \leftarrow \textbf{if } s = \Lambda \textbf{ then } hd(t) \textbf{ else } hd(s).$$

Observe that this program will produce an error if $s = t = \Lambda$. But since conventional evaluation of $hd(s \circ t)$ will produce the same error, this expression procedure is correct in this case.

These examples show how the composition rule can be thought of as a context specification rule. In these examples, however, none of the resulting programs were recursive. We will consider this case in the next two sections.

2. Specialization of Recursive Functions

Let us consider first an example, involving the familiar *rev* function,

$$rev(x) \leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(x)) \circ cons(hd(x), \Lambda).$$

Suppose that in some context in which *rev* is called we require only the first element of the resulting list. That is, we need to compute $hd(rev(x))$ for some x. (We will assume throughout that $x \neq \Lambda$.)

To describe this situation, we can write

Compute: $hd(rev(x))$ (where $x \neq \Lambda$)
Given: $rev(x) \leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(x)) \circ cons(hd(x), \Lambda)$,

as in Chapter 1. It seems natural that we should be able to develop a *specialized* version of *rev* for this particular application, in which computation not necessary to the determination of the *hd* of the result list is eliminated. As expected, we do this by forming an expression procedure for $hd(rev(x))$,

$$hd(rev(x)) \leftarrow hd\big(\textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(x)) \circ cons(hd(x), \Lambda)\big).$$

Our goal in simplifying this expression procedure is to eliminate the recursive call or, failing this, to bring it into the specialized form. The advantages

of eliminating the call are obvious; if we can bring the call into the specialized format, then we have a potential gain in efficiency at all levels of recursion, rather than just at the first. That is, the efficiency improvement that results from the additional contextual information provided by the composition operation can be realized throughout the recursive computation.

Some significant amount of simplification yields

$$hd(rev(x)) \leftarrow \textbf{if } tl(x) = \Lambda \textbf{ then } hd(x) \textbf{ else } hd(rev(tl(x))).$$

(This example is worked out in detail later in this chapter.) We see that the recursive call is now in the specialized form; thus, it is possible to compute $hd(rev(x))$ without ever calling *rev*.

In general, a basic evaluator can be implemented more efficiently than a full evaluator for programs with expression procedures, since the full evaluator requires a pattern matching capability. Therefore, we generally seek to transform recursive expression procedures into equivalent recursive basic definitions. We do this in two steps, first by abstracting,

$$hd(rev(x)) \leftarrow hdrev(x)$$
$$hdrev(x) \leftarrow \textbf{if } tl(x) = \Lambda \textbf{ then } hd(x) \textbf{ else } hd(rev(tl(x))),$$

and then applying, both in the body,

$$hdrev(x) \leftarrow \textbf{if } tl(x) = \Lambda \textbf{ then } hd(x) \textbf{ else } hdrev(tl(x)),$$

and in the original specialized call. We thus obtain the basic program

Compute: $hdrev(z)$ (where $z \neq \Lambda$)
Given: $hdrev(x) \leftarrow \textbf{if } tl(x) = \Lambda \textbf{ then } hd(x) \textbf{ else } hdrev(tl(x))$.

Observe that *cons* is never called in this program, so we have realized a significant improvement in efficiency as a result of this specialization.

To summarize, the specialization process has three steps. Suppose we are given a definition of a function f that has recursive calls labeled f_1, f_2, ..., and suppose that f is called in a specialized context, $h(f)$.

Compute: $h(f)$
Given: $f \leftarrow \cdots f_1 \cdots f_2 \cdots$

We perform the following steps:

(1) (*Compose*) Compose the definition of f into an expression procedure for $h(f)$.

(2) (*Simplify*) Simplify this expression procedure until each recursive call in the body either has the specialized form, or has been eliminated. If this fails, then *exit*.

(3) (*Rename*) Abstract and apply, in effect renaming $h(f)$ to hf.

Compute: hf
Given: $hf \leftarrow \cdots hf_1 \cdots hf_2 \cdots$

(The examples in the first section of this chapter are all examples of specializations in which the recursive calls are all eliminated in the simplification step.)

In practice, we usually specialize by degrees, in order to obtain the best possible improvement to a program. For example, if the compute specification above had been,

Compute: $h_1(h_2(f))$,

then we would have attempted specialization first on $h_2(f)$. If this specialization had succeeded, then we would have attempted to specialize the resulting function $h_2 f$ to $h_1(h_2 f)$. Now if this second specialization fails, then we would at least realize the benefits of the first specialization. Note that if the *first* specialization had failed, then it is still possible for a full specialization of f to $h_1(h_2(f))$ to succeed, since the additional assumptions may permit more simplification of the expression procedure body.

It is apparent that many program derivation examples take this form. The improvement we realized in the initial *rev* example could be considered to be the result of specializing the *recursive instance of rev*. We can demonstrate this by deriving the efficient list reversing function in a manner more faithful to the specialization scheme. We start by taking the initial definition of *rev*,

$$rev(x) \leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(x)) \circ cons(hd(x), \Lambda),$$

abstracting the recursive call,

$$rev(x) \leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } rev0(tl(x)) \circ cons(hd(x), \Lambda)$$
$$rev0(x) \leftarrow rev(x),$$

and applying the definition of *rev* in the body of *rev*0,

$$rev(x) \leftarrow \text{if } x = \Lambda \text{ then } \Lambda \text{ else } rev0(tl(x)) \circ cons(hd(x), \Lambda)$$
$$rev0(x) \leftarrow \text{if } x = \Lambda \text{ then } \Lambda \text{ else } rev0(tl(x)) \circ cons(hd(x), \Lambda).$$

We can now view the program improvement process as one in which the auxiliary function *rev*0 is specialized to $rev0(u) \circ v$ for lists u and v. After composing and simplifying, we obtain

$$rev0(u) \circ v \leftarrow \text{if } u = \Lambda \text{ then } v \text{ else } rev0(tl(u)) \circ cons(hd(u), v).$$

The specialization is successful, so we rename the complex name $rev0(u) \circ v$ to a new basic name $rev1(u, v)$ and eliminate the now useless *rev*0.

$$rev(x) \leftarrow \text{if } u = \Lambda \text{ then } v \text{ else } rev1(tl(u), cons(hd(u), v))$$
$$rev1(u, v) \leftarrow \text{if } u = \Lambda \text{ then } v \text{ else } rev1(tl(u), cons(hd(u), v))$$

To obtain the program we derived in Chapter 1, we abstract the bodies of both definitions into a new function $rev2(u, v)$,

$$rev(x) \leftarrow rev2(x, \Lambda)$$
$$rev1(u, v) \leftarrow rev2(u, v)$$
$$rev2(u, v) \leftarrow \text{if } u = \Lambda \text{ then } v \text{ else } rev1(tl(u), cons(hd(u), v)),$$

and unfold the single *rev*1 call and eliminate *rev*1, giving

$$rev(x) \leftarrow rev2(x, \Lambda)$$
$$rev2(u, v) \leftarrow \text{if } u = \Lambda \text{ then } v \text{ else } rev2(tl(u), cons(hd(u), v)).$$

3. Boundary Shifting

In this section, we give an intuitive explanation of how the use of expression procedures enables us to improve recursive programs.

Our ability to improve a program depends to a great extent on our capability of passing information across procedure boundaries. The boundary

represented by a basic name in a program is a particularly narrow one, in the sense that only arguments, results, and the procedure name itself are shared on both sides of the boundary.

In fact, this was one of the insights represented in the inductive-assertions method — in which the narrow boundaries represented by procedure names or loop entry points are explicitly annotated by assertions, called *invariants*, which specify relationships across the boundary. Determination of such invariants for the purposes of program proof or manipulation may be quite difficult. (The inductive proof methods we introduce later in this chapter are based on this approach.)

An expression procedure name, which is a complex expression, provides more information on the two sides of the boundary it specifies than does an ordinary basic name. Indeed, an instance of an expression procedure name is an expression that can be evaluated even without reference to the expression procedure itself.

Our use of the three program tranformation rules can thus be thought of in terms of the broadening and shifting of boundaries in recursive programs. For example, in the *rev* example, we started with the definition

$$rev(x) \leftarrow \text{if } x = \Lambda \text{ then } \Lambda \text{ else } rev(tl(x)) \circ cons(hd(x), \Lambda).$$

We improved this program as the result of our use of the associative property of the *append* function across the procedure boundary — changing the association between the call to *append* in the body of *rev* and the call occurring in the body of the recursive call. We did this by widening the boundary to include the call:

$$rev(u) \circ v \leftarrow \text{if } u = \Lambda \text{ then } \Lambda \circ v \text{ else } \big(rev(tl(u)) \circ cons(hd(u), \Lambda)\big) \circ v,$$

and then associating to the right,

$$rev(u) \circ v \leftarrow \text{if } u = \Lambda \text{ then } \Lambda \circ v \text{ else } rev(tl(u)) \circ \big(cons(hd(u), \Lambda) \circ v\big).$$

Recall that computation of $a \circ b$ requires at least $|a|$ calls to *cons* (where $|a|$ is the length of the list a). Thus, computation of $(a \circ b) \circ c$ requires $|a| + |a \circ b| = 2|a| + |b|$ calls to *cons*. By associating to the right, however, we obtain $a \circ (b \circ c)$, which requires exactly $|a| + |b|$ *cons* calls.

The renaming steps that followed had the effect of replacing the wide boundary "$rev(u) \circ v$" by a narrow, and thus computationally more efficient boundary "$rev2$."

4. Two Examples

We now give several examples of derivations to illustrate the remarks above on specialization and boundary shifting.

Fibonacci. In the first example, we transform an inherently exponential Fibonacci program into a linear one. This example also illustrates our technique for handling multiple argument functions.

In this derivation, we make use of n-tuples as a general grouping device in order to permit a more uniform treatment of functions with multiple arguments and results. The idea is to consider all n-ary functions as being, in fact, functions of a single argument that is an n-tuple. This allows, for example, direct composition of $f_1: A \times B \mapsto C \times D$ and $f_2: C \times D \mapsto E$ into $f_1 \circ f_2: A \times B \mapsto E$. We often indicate the use of groups by writing function applications such as $g(u, v)$ as $g(\langle u, v \rangle)$. (Thus, the only true n-ary function is the tuple formation function.)

Grouping gives added flexibility to the transformation rules, as we shall see below.

We start with a program based directly on the definition of the Fibonacci function,

$$f(x) \leftarrow \textbf{if } x \leq 1 \textbf{ then } x \textbf{ else } f(x-1) + f(x-2).$$

Here we assume that all functions are primitive except f and that x is a nonnegative integer. In the computation of $f(x)$ using this definition, the value of $f(x-i)$, where $x > i$, is computed exactly $f(i)$ times. Since $\sum_{1 \leq i < x} f(i)$ is exponential in x, the total amount of computation is also exponential in x.

Using the grouping notation, this definition can be written as

$$f(x) \leftarrow \textbf{if } x \leq 1 \textbf{ then } x \textbf{ else } plus(\langle f(x-1), f(x-2) \rangle).$$

This tempts us to consider the pair $\langle f(x-1), f(x-2) \rangle$ as a new boundary for f, and to attempt specialization based on this. The first step of specialization is composition: We compose on the first element of the pair to obtain the definition,

$$\langle f(x-1), f(x-2) \rangle \leftarrow \langle \textbf{if } x \leq 2 \textbf{ then } x \textbf{ else } plus(\langle f(x-2), f(x-3) \rangle), f(x-2) \rangle.$$

This definition is simplified in two steps. First, we distribute the pairing over the conditional to obtain

$$\langle f(x-1), f(x-2) \rangle \leftarrow \textbf{if } x \leq 2 \textbf{ then } \langle x, f(x-2) \rangle \textbf{ else } \langle plus(\langle f(x-2), f(x-3) \rangle), f(x-2) \rangle.$$

Second, we unfold the instance of $f(x-2)$ in the then clause, and simplify the result based on the condition $x \leq 2$, obtaining

$$\langle f(x-1), f(x-2) \rangle \leftarrow \textbf{if } x \leq 2 \textbf{ then } \langle x-1, x-2 \rangle \textbf{ else } \langle plus(\langle f(x-2), f(x-3) \rangle), f(x-2) \rangle.$$

(We describe in the next section a general mechanism for making simplifications based on contextual conditions.)

Since the term $f(x-2)$ occurs twice in the else clause, it is natural to abstract part of the else clause. If we are clever about this, then a recursive call to the expression procedure could be formed, and our specialization will be successful:

$$\langle f(x-1), f(x-2) \rangle \leftarrow \textbf{if } x \leq 2 \textbf{ then } \langle x-1, x-2 \rangle \textbf{ else } h(\langle f(x-2), f(x-3) \rangle)$$
$$h(\langle u, v \rangle) \leftarrow \langle plus(\langle u, v \rangle), u \rangle.$$

(In an implemented system, it may be best for the simplification mechanism to operate in a goal-directed manner, attempting to eliminate or specialise recursive calls.)

With this recursive expression procedure, f can be computed in time linear in the value of x. It remains to rename boundaries, eliminating the complex boundary in favor of a new narrow one. We first abstract the body

of the expression procedure,

$$\langle f(x-1), f(x-2)\rangle \leftarrow g(x)$$
$$g(x) \leftarrow \textbf{if } x \leq 2 \textbf{ then } \langle x-1, x-2\rangle \textbf{ else } h\big(\langle f(x-2), f(x-3)\rangle\big),$$

and then unfold the expression procedure call in the **else** clause to obtain

$$g(x) \leftarrow \textbf{if } x \leq 2 \textbf{ then } \langle x-1, x-2\rangle \textbf{ else } h(g(x-1)).$$

Only one instance of the expression procedure name remains, which is the original call in the definition of f. We unfold it, and eliminate the now useless expression procedure to get the final program,

$$f(x) \leftarrow \textbf{if } x \leq 1 \textbf{ then } 1 \textbf{ else } plus(g(x))$$
$$g(x) \leftarrow \textbf{if } x \leq 2 \textbf{ then } \langle x-1, x-2\rangle \textbf{ else } h(g(x-1))$$
$$h(\langle u, v\rangle) \leftarrow \langle u+v, u\rangle.$$

How did the efficiency improvement come about? Considering the two recursive calls in f as a single complex boundary gave us a way of merging redundant computation in the two recursive instances. It turned out that this merging resulted in a single recursive instance of this same new boundary, rather than in four more recursive calls to f.

(Minsky suggested this example in [Minsky70]. A similar linear Fibonacci program was derived by Burstall and Darlington in [Burstall77]. We will extensively compare their approach and ours in a later section.)

3-ary append. We consider now a program for computing $(a \circ b) \circ c$, where a, b, and c are lists. If the *append* function is defined as

$$s \circ t \leftarrow \textbf{if } s = \Lambda \textbf{ then } t \textbf{ else } cons(hd(s), tl(s) \circ t),$$

then (as we noted earlier) direct computation of $(a \circ b) \circ c$ is inefficient, since the total number of calls to *cons* will be equal to the length of a plus the length of $a \circ b$, which is equal to the length of b plus twice the length of a. Since we can prove that

$$(a \circ b) \circ c = a \circ (b \circ c),$$

and $a \circ (b \circ c)$ requires a number of calls to *cons* equal only to the length of a plus the length of b, then we know there is a more efficient way of computing $(a \circ b) \circ c$.

We can view this as a specialization problem, our goal being to specialise *append* to the case of computing $(a \circ b) \circ c$.

We use ordinary composition on the inner call to *append* to obtain the definition

$$(a \circ b) \circ c \leftarrow \big(\textbf{if } a = \Lambda \textbf{ then } b \textbf{ else } cons(hd(a), tl(a) \circ b)\big) \circ c.$$

Distributing in the outer *append* call results in

$$(a \circ b) \circ c \leftarrow \textbf{if } a = \Lambda \textbf{ then } b \circ c$$
$$\textbf{else } cons(hd(a), tl(a) \circ b) \circ c.$$

Now we unfold the outer call to *append* in the **else** clause. Since the **if** test of the outer *append* is always false, this simplifies to

$$(a \circ b) \circ c \leftarrow \textbf{if } a = \Lambda \textbf{ then } b \circ c$$
$$\textbf{else } cons(hd[cons(hd(a), tl(a) \circ b)],$$
$$tl[cons(hd(a), tl(a) \circ b)] \circ c),$$

which, on the basis of properties of *hd*, *tl*, and *cons*, further simplifies to

$$(a \circ b) \circ c \leftarrow \textbf{if } a = \Lambda \textbf{ then } b \circ c$$
$$\textbf{else } cons\big(hd(a), (tl(a) \circ b) \circ c\big),$$

which contains a recursive call.

Thus, our specialization is successful, as we abstract and apply to obtain the final pair of definitions,

$$(a \circ b) \circ c \leftarrow a3(a, b, c)$$
$$a3(a, b, c) \leftarrow \textbf{if } a = \Lambda \textbf{ then } b \circ c \textbf{ else } cons\big(hd(a), a3(tl(a), b, c)\big).$$

(This idea for this example came from a similar derivation by Wegbreit [Wegbreit76]. We will compare his method and ours in a later section. Chatelin also presents a derivation of this function [Chatelin77].)

5. Conditional Definitions: Qualifiers

Thus far, we have created specialized versions of a given definition by making expression procedures for new expressions that contain the given name. The specialization condition is thus expressed as a constraint on the syntactic context of instances of the given name. It is often the case that we need to impose a specialization condition that cannot be expressed in this syntactic way. In such cases, we specify the condition by means of special condition tags, called *qualifiers*.

In general, a definition can have associated with it a ***qualifier***, which is an expression that evaluates to a truth value. If a definition has a qualifier, then the definition can be expanded by an evaluator or by the application transformation only in contexts in which the qualifier is true. Thus, we can think of qualifiers as specifying preconditions for definitions. (This is similar to Wegbreit's notion of *context* tags [Wegbreit76].)

We write a qualified definition as

$$\langle P \rangle \quad E \leftarrow F,$$

where E and F are expressions and P is a primitive expression such that the variables of P are a subset of the variables of E.

For example, our program for appending an empty list could have been written as

$$\langle s{=}\Lambda \rangle \quad s \circ t \leftarrow t.$$

The qualifier implies that this expression procedure can be applied or evaluated only when the condition specified in the qualifier is true, namely that $s = \Lambda$. That is, the expression procedure denotes a conditional equivalence, that $s \circ t$ and t are equivalent if $s = \Lambda$.

To obtain the above expression procedure, we started with the definition of ***append***,

$$s \circ t \leftarrow \textbf{if } s = \Lambda \textbf{ then } t \textbf{ else } cons(hd(s), tl(s) \circ t),$$

and introduced a new definition that was created by adding the qualifier to this definition:

$$\langle s{=}\Lambda \rangle \quad s \circ t \leftarrow \textbf{if } s = \Lambda \textbf{ then } t \textbf{ else } cons(hd(s), tl(s) \circ t).$$

In the same way that we can simplify the branches of a conditional based on the truth value of the condition, we can simplify the entire right-hand side of a definition based on the qualifiers associated with the definition. Since the **if** condition on the right-hand side, $s = \Lambda$, simplifies to **true** in the presence of the qualifier, $\langle s{=}\Lambda \rangle$, the entire conditional can be replaced by its **then** clause, giving

$$\langle s{=}\Lambda \rangle \quad s \circ t \leftarrow t.$$

In the $hd(s \circ t)$ example, we could have introduced qualifiers to limit use of the expression procedure to the cases in which the ***hd*** is defined, giving

$$\langle s{\neq}\Lambda \vee t{\neq}\Lambda \rangle \quad hd(s \circ t) \leftarrow \textbf{if } s = \Lambda \textbf{ then } hd(t) \textbf{ else } hd(s).$$

Strictly speaking, however, the qualifiers are unnecessary since if $s = t = \Lambda$, then $hd(s \circ t)$ will produce the same error whether evaluated conventionally or with the expression procedure.

Definitions with qualifiers are evaluated just like other definitions, except that the qualifiers are tested first. Therefore, if we want to use the application rule to unfold a qualified definition, we must have some way of proving that the qualifier will always be true in the given context. To facilitate these proofs, we admit the qualifier notation within definition bodies to remind us of conditions that are true. These "qualifiers," are called ***body qualifiers*** or ***tags***, and may appear anywhere in a definition body. Unlike ordinary qualifiers, which denote preconditions on the applicability of definitions, and which must be checked at run-time, tags in definition bodies denote only assertions, and are ignored by the evaluator. Thus, we place no restrictions on the form of the expressions they contain.

Qualified definitions are introduced to a program by an extension of the composition rule: If A and F are expressions, and P is a primitive expression denoting a truth value, such that all free variables in P are among the free variables of A, then

$$\textbf{given} \quad A \leftarrow F, \qquad \textbf{add } \langle P \rangle\, A \leftarrow \langle P \rangle\, F.$$

That is, if $A \leftarrow F$ is a definition then for a given legal qualifier $\langle P \rangle$ we can create a definition of A conditional on P.

The body qualifier indicates only that F may be simplified based on the assumption that P is true. Usually we omit the body qualifier since it is obvious from context.

Because our correct use of the application rule for qualified definitions depends on the use of body qualifiers, it is necessary that we explicitly specify rules for their use. Example of such rules are,

$$\begin{aligned} \{P\}\ \textbf{if } Q \textbf{ then } R \textbf{ else } S &= \textbf{if } Q \textbf{ then } \{P\}\ R \textbf{ else } \{P\}\ S \\ \textbf{if } Q \textbf{ then } R \textbf{ else } S &= \textbf{if } Q \textbf{ then } \{Q\}\ R \textbf{ else } \{\neg Q\}\ S \\ \textbf{if } Q \textbf{ then } \{R'\}\ R \textbf{ else } \{S'\}\ S &= \{\textbf{if } Q \textbf{ then } R' \textbf{ else } S'\} \\ &\qquad \textbf{if } Q \textbf{ then } R \textbf{ else } S. \end{aligned}$$

Also, if a body qualifier holds for a given term, then it holds for every subterm of that term. Body qualifiers can, of course, be dropped from an expression at any time. (In general, special rules like the ones for **if-then-else** above must be given for each nonstrict primitive name.)

For example, given the definition

$$\{list(s) \wedge list(t)\} \quad s \circ t \leftarrow \textbf{if } s = \Lambda \textbf{ then } t \textbf{ else } cons(hd(s), tl(s) \circ t),$$

we can apply the body qualifier rules to obtain

$$\{list(s) \wedge list(t)\}\ s \circ t \leftarrow \textbf{if } s = \Lambda \textbf{ then } t \textbf{ else } \{list(s) \wedge list(t) \wedge s \neq \Lambda\}\ cons(hd(s), tl(s) \circ t).$$

Since $list(s) \wedge s \neq \Lambda$ implies $list(tl(s))$, we have

$$\{list(s) \wedge list(t)\}\ s \circ t \leftarrow \textbf{if } s = \Lambda \textbf{ then } t \textbf{ else } \{list(tl(s)) \wedge list(t)\}\ cons(hd(s), tl(s) \circ t),$$

thus establishing that the definition is recursive.

6. Qualifiers: Three Examples

We now give several detailed examples to illustrate the use of qualifiers in program manipulation. The first two examples illustrate relatively simple applications of qualifiers to the improvement of list manipulating programs. In the third example, we improve an integer square root program.

Last. In the first example, we improve a function for computing the last element of a list:

$$\begin{aligned} last(x) \leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \\ \textbf{elseif } tl(x) = \Lambda \textbf{ then } hd(x) \\ \textbf{else } last(tl(x)). \end{aligned}$$

We seek to improve this definition based on the following observation: The initial **if** test in the body of this definition can only be true on the very first iteration. This is true because we know that because of the **elseif** condition the argument to *last* on all of its recursive calls is a nonempty list:

$$\begin{aligned} last(x) \leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \\ \textbf{elseif } tl(x) = \Lambda \textbf{ then } hd(x) \\ \textbf{else } \{tl(x) \neq \Lambda\}\ last(tl(x)), \end{aligned}$$

We start by developing a specialized version of *last* for this case. Because of the body qualifier, we know that this specialization will trivially succeed: Using the composition rule, we get

$$\begin{aligned} \{x \neq \Lambda\}\ last(x) \leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \\ \textbf{elseif } tl(x) = \Lambda \textbf{ then } hd(x) \\ \textbf{else } \{tl(x) \neq \Lambda\}\ last(tl(x)), \end{aligned}$$

which simplifies to

$$\begin{aligned} \{x \neq \Lambda\}\ last(x) \leftarrow \textbf{if } tl(x) = \Lambda \textbf{ then } hd(x) \\ \textbf{else } \{tl(x) \neq \Lambda\}\ last(tl(x)), \end{aligned}$$

and is recursive.

The next step of specialization involves abstracting the body of this definition. We observe, however, that the unqualified definition of *last* contains this same definition body as a subexpression, so we can abstract both simultaneously. We obtain

$$\begin{aligned} last(z) &\leftarrow \textbf{if } z = \Lambda \textbf{ then } \Lambda \textbf{ else } last2(z) \\ \{z \neq \Lambda\}\ last(z) &\leftarrow last2(z) \\ last2(z) &\leftarrow \textbf{if } tl(z) = \Lambda \textbf{ then } hd(z) \\ &\qquad \textbf{else } \{tl(z) \neq \Lambda\}\ last(tl(z)). \end{aligned}$$

Unfolding the expression procedure call, and eliminating the now useless expression procedure, yields the program

$$\begin{aligned} last(z) &\leftarrow \textbf{if } z = \Lambda \textbf{ then } \Lambda \textbf{ else } last2(z) \\ last2(z) &\leftarrow \textbf{if } tl(z) = \Lambda \textbf{ then } hd(z) \textbf{ else } last2(tl(z)). \end{aligned}$$

There is yet another improvement we can make to this program. We observe that $tl(z)$ is computed twice on each iteration. To eliminate this common subexpression, we "rotate" the definition of *last2*: First, we abstract the body of *last2*,

$$\begin{aligned} last2(z) &\leftarrow lasta(z, tl(z)) \\ lasta(z, u) &\leftarrow \textbf{if } u = \Lambda \textbf{ then } hd(z) \textbf{ else } last2(u); \end{aligned}$$

then we unfold the two calls to *last2* to obtain the final program,

$$\begin{aligned} last(z) &\leftarrow \textbf{if } z = \Lambda \textbf{ then } \Lambda \textbf{ else } lasta(z, tl(z)) \\ lasta(z, u) &\leftarrow \textbf{if } u = \Lambda \textbf{ then } hd(z) \textbf{ else } lasta(u, tl(u)). \end{aligned}$$

This is an optimal implementation of the *last* function over the primitives *cons*, *hd*, and *tl*.

Hdrev. The second example starts with the list reversing function, *rev*.

$$rev(z) \leftarrow \textbf{if } z = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(z)) \circ cons(hd(z), \Lambda)$$

As promised in an earlier section, we develop here in detail a specialized version of *rev* for the case in which we require only the first element of the result; that is, for $\{z \neq \Lambda\}\ hd(rev(z))$.

We assume throughout this derivation that all function arguments are lists, so we will not introduce explicit qualifiers to indicate this.

We will need some subsidiary functions for this derivation. One of those,

$$hd(s \circ t) \leftarrow \textbf{if } s = \Lambda \textbf{ then } hd(t) \textbf{ else } hd(s),$$

we derived for the *append* function in a previous section. There are two others that we derive now.

The first is $null(a \circ b)$, where a and b are lists. (We use the notation $null(a)$ as an abbreviation for $a = \Lambda$.) Composing with the *append* function, we obtain

$$null(s \circ t) \leftarrow null(\textbf{if } s = \Lambda \textbf{ then } t \textbf{ else } cons(hd(s), tl(s) \circ t)).$$

Since *append* terminates for all lists, this simplifies directly to

$$null(s \circ t) \leftarrow \textbf{if } s = \Lambda \textbf{ then } null(t) \textbf{ else false},$$

or

$$null(s \circ t) \leftarrow \textbf{if } null(s) \textbf{ then } null(t) \textbf{ else false}.$$

The final subsidiary function is for $null(rev(z))$, where z is a list. First, we compose to get

$$null(rev(z)) \leftarrow null(\textbf{if } z = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(z)) \circ cons(hd(z), \Lambda)).$$

We first bring the calls to *null* inside,

$$null(rev(z)) \leftarrow \textbf{if } z = \Lambda \textbf{ then true else } null(rev(tl(z)) \circ cons(hd(z), \Lambda));$$

then we apply the expression procedure we just derived for $null(s \circ t)$ to the else clause to obtain

$$\begin{aligned} null(rev(z)) \leftarrow\ & \textbf{if } z = \Lambda \textbf{ then true} \\ & \textbf{elseif } null(rev(tl(z))) \textbf{ then } null(cons(hd(z), \Lambda)) \\ & \textbf{else false}. \end{aligned}$$

(The expression procedure for $null(s \circ t)$ is not strict in t, so before applying it, we needed to prove that evaluation of $rev(tl(z))$ terminates.)

Since $null(cons(hd(z), \Lambda))$ is always false, this simplifies to

$$null(rev(z)) \leftarrow \textbf{if } z = \Lambda \textbf{ then true else false},$$

or

$$null(rev(z)) \leftarrow null(z).$$

Given these auxiliary definitions, the derivation for $hd(rev(z))$ becomes quite straightforward. Starting with the basic definition of rev, we compose to obtain the expression procedure,

$$(z \neq \Lambda)\ hd(rev(z)) \leftarrow hd\big(\textbf{if } z = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(z)) \circ cons(hd(z), \Lambda)\big).$$

The qualifier enables us to simplify the conditional, replacing it by its **else** clause:

$$(z \neq \Lambda)\ hd(rev(z)) \leftarrow hd\big(rev(tl(z)) \circ cons(hd(z), \Lambda)\big).$$

We now apply the definition we derived above for $hd(s \circ t)$ to get

$$(z \neq \Lambda)\ hd(rev(z)) \leftarrow \textbf{if } null(rev(tl(z))) \textbf{ then } hd(cons(hd(z), \Lambda)) \textbf{ else } hd(rev(tl(z))).$$

The **then** clause can be simplified, and the definition for $null(rev(z))$ that we derived above can be applied to the **if** test, giving

$$(z \neq \Lambda)\ hd(rev(z)) \leftarrow \textbf{if } null(tl(z)) \textbf{ then } hd(z) \textbf{ else } (tl(z) \neq \Lambda)\ hd(rev(tl(z))),$$

which is recursive.

Thus, our specialization is successful, and it remains only to rename. Abstraction yields the pair of definitions

$$(z \neq \Lambda)\ hd(rev(z)) \leftarrow hdrev(z)$$
$$hdrev(z) \leftarrow \textbf{if } null(tl(z)) \textbf{ then } hd(z) \textbf{ else } (tl(z) \neq \Lambda)\ hd(rev(tl(z))).$$

Unfolding the expression procedure call gives the final set of definitions

$$(z \neq \Lambda)\ hd(rev(z)) \leftarrow hdrev(z)$$
$$hdrev(z) \leftarrow \textbf{if } null(tl(z)) \textbf{ then } hd(z) \textbf{ else } hdrev(tl(z)).$$

Note that this definition of *hdrev* is identical to that of the *last2* function derived in the previous example, thus proving that the first element of a reversed list is the same as the last element of the initial list. As in the previous derivation, we can continue our transformations to avoid computing $tl(z)$ twice. After abstracting and applying, we obtain

$$(z \neq \Lambda)\ hd(rev(z)) \leftarrow hrev(z, tl(z))$$
$$hrev(z, u) \leftarrow \textbf{if } null(u) \textbf{ then } hd(z) \textbf{ else } hrev(u, tl(u)).$$

Square root. The final example involves improving the following program for computing the integer square root of a nonnegative integer.

$$s(z) \leftarrow q(0, z)$$
$$q(i, z) \leftarrow \textbf{if } i^2 \leq z < (i+1)^2 \textbf{ then } i \textbf{ else } q(i+1, z)$$

We will assume that the domain of all variables is the set of nonnegative integers, and so we will not introduce explicit "domain" qualifiers. This program finds the square root of z by a "brute force" algorithm, checking successive nonnegative integers i until a value is found such that z is in the interval $[i^2, (i+1)^2)$. The program is inefficient because it squares the current values of i and $i+1$ on each iteration. We develop a more efficient version by specializing q in several steps.

The first observation we make concerns the **if** test in the definition of q. It appears that the calls to q are arranged such that $i^2 < z$ is always true before $q(i, z)$ is called. If this is indeed true, then half of the **if** test is unnecessary. We can test this hypothesis for specialization by composing q with a qualifier:

$$(i^2 \leq z)\ q(i, z) \leftarrow \textbf{if } (i^2 \leq z) \wedge (z < (i+1)^2) \textbf{ then } i \textbf{ else } q(i+1, z)$$

The qualifier implies that the first conjunct in the **if** test is always true, so that conjunct can be eliminated:

$$(i^2 \leq z)\ q(i, z) \leftarrow \textbf{if } z < (i+1)^2 \textbf{ then } i \textbf{ else } q(i+1, z)$$

For our specialization to succeed, we must show that all calls to q are in the specialized form. Since the negation of the **if** test,

$$not(x < (i+1)^2) \quad = \quad (i+1)^2 \leq x,$$

is true in the **else** clause, we see that the expression procedure is recursive. In addition, since x is assumed to be a nonnegative integer, it is clear that the qualifier condition is true on the initial call of q from s. We have

$$\begin{aligned} s(x) &\leftarrow \quad (0^2 < x)\ q(0,x) \\ (i^2 < x)\ q(i,x) &\leftarrow \textbf{ if } x < (i+1)^2 \textbf{ then } i \textbf{ else } \quad ((i+1)^2 \leq x)\ q(i+1,x). \end{aligned}$$

Therefore, our specialization is successful.

Before renaming, we consider improving the program further, based on the following additional observation: Since $(i+1)^2 = i^2 + 2i + 1$, the value of $(i+1)^2$ can be obtained from the value of i^2 simply by adding the quantity $2i+1$, thus trading a multiplication for three additions.

To carry over this old value of i^2, we abstract q in an unusual way:

$$\begin{aligned} (i^2 \leq x)\ q(i,x) &\leftarrow h(i,i^2,x) \\ h(i,j,x) &\leftarrow \textbf{if } x < (i+1)^2 \textbf{ then } i \textbf{ else } \quad ((i+1)^2 \leq x)\ q(i+1,x), \end{aligned}$$

defining a new symbol h that has three parameters, i, j, and x. The new parameter j corresponds to the value of i^2, so $(i+1)^2$ can be computed in the body of h using this value. Unfortunately, it appears that we cannot simplify the body of h without an explicit assertion that $i^2 = j$, since h can take arbitrary values of i and j. We observe, however, that the required property holds at the only call to h, so we can always use the following specialized version of h:

$$(i^2 = j)\ h(i,j,x) \leftarrow \textbf{if } x < (i+1)^2 \textbf{ then } i \textbf{ else } \quad ((i+1)^2 \leq x)\ q(i+1,x)$$

Our program now has the following form:

$$\begin{aligned} s(x) &\leftarrow \quad (0^2 \leq x)\ q(0,x) \\ (i^2 \leq x)\ q(i,x) &\leftarrow \quad (i^2 = (i^2))\ h(i,i^2,x) \\ (i^2 = j)\ h(i,j,x) &\leftarrow \textbf{if } x < (i+1)^2 \textbf{ then } i \textbf{ else } \quad ((i+1)^2 \leq x)\ q(i+1,x). \end{aligned}$$

We can now apply to eliminate q in favor of h, thus completing the original specialization process for q:

$$\begin{aligned} s(x) &\leftarrow \quad (0^2 = (0^2))\ h(0,0,x) \\ (i^2 = j)\ h(i,j,x) &\leftarrow \textbf{ if } x < (i+1)^2 \textbf{ then } i \\ &\qquad \textbf{else } \quad ((i+1)^2 = ((i+1)^2))\ h(i+1,(i+1)^2,x). \end{aligned}$$

The new qualifier enables us to simplify the $(i+1)^2$ term,

$$\begin{aligned} (i^2 = j)\ h(i,j,x) &\leftarrow \textbf{ if } x < j + 2i + 1 \textbf{ then } i \\ &\qquad \textbf{else } \quad ((i+1)^2 = j + 2i + 1)\ h(i+1, j+2i+1, x), \end{aligned}$$

and then to rename, eliminating h and its associated run-time qualifier test. We first abstract, to obtain,

$$\begin{aligned} (i^2 = j)\ h(i,j,x) &\leftarrow k(i,j,x) \\ k(i,j,x) &\leftarrow \textbf{ if } x < j + 2i + 1 \textbf{ then } i \\ &\qquad \textbf{else } \quad ((i+1)^2 = j + 2i + 1)\ h(i+1, j+2i+1, x), \end{aligned}$$

and then unfold the two calls to h, yielding the basic program,

$$\begin{aligned} s(x) &\leftarrow k(0,0,x) \\ k(i,j,x) &\leftarrow \textbf{ if } x < j + 2i + 1 \textbf{ then } i \textbf{ else } k(i+1, j+2i+1, x). \end{aligned}$$

This program does no multiplications, but it does require seven additions to be performed at each step of recursion. We can improve this by abstracting to eliminate the redundant computation of $j + 2i + 1$:

$$\begin{aligned} s(x) &\leftarrow k(0,0,x) \\ k(i,j,x) &\leftarrow p(i, j+2i+1, x) \\ p(i,m,x) &\leftarrow \textbf{if } x < m \textbf{ then } i \textbf{ else } k(i+1,m,x). \end{aligned}$$

Unfolding eliminates k in favor of p, yielding

$$\begin{aligned} s(x) &\leftarrow p(0,1,x) \\ p(i,m,x) &\leftarrow \textbf{if } x < m \textbf{ then } i \textbf{ else } p(i+1, m+2i+3, x), \end{aligned}$$

which does only four additions at each step.

As a final improvement, we can abstract again to eliminate one of these additions. We abstract the body of p and, as above, add a new parameter and an associated qualifier:

$$p(i,m,z) \leftarrow \ \{2i+3=(2i+3)\}\ t(i,m,2i+3,z)$$
$$\{2i+3=n\}\ t(i,m,n,z) \leftarrow \textbf{if } z < m \textbf{ then } i \textbf{ else } p(i+1, m+2i+3, z).$$

After eliminating p and renaming to eliminate the qualified definition, we obtain our final iterative program for integer square root,

$$s(z) \leftarrow r(0,1,3,z)$$
$$r(i,m,n,z) \leftarrow \textbf{if } z < m \textbf{ then } i \textbf{ else } r(i+1, m+n, n+2, z),$$

which requires only three additions at each step of recursion.

(A similar derivation appears in [Katz78].)

7. A Note on Qualifiers and Safety

One of the uses of qualifiers is in ensuring the "safety" of various partial functions. We describe this by means of an example: Suppose we admit a distinguished value, error, to our domain to indicate an undefined value. An integer division procedure may thus be written

$$idiv(x,y) \leftarrow \textbf{if } y = 0 \textbf{ then error else } E(x,y),$$

where E is some expression that actually calculates the quotient of x and y assuming that $y \neq 0$.

Now, if we can prove in the context of a given call to *idiv* that $y \neq 0$, then we would like to use a version of *idiv* specialized to this error-free case. Using composition, we introduce the expression procedure,

$$\{y\neq 0\}\ idiv(x,y) \leftarrow \textbf{if } y = 0 \textbf{ then error else } E(x,y),$$

which, by virtue of the qualifier, immediately simplifies to

$$\{y\neq 0\}\ idiv(x,y) \leftarrow E(x,y),$$

thus eliminating the test.

It appears, however, that in order to take advantage of this efficient direct call to E, we must either implement a capability for expression procedures in a run-time system, or "open code" the body of *idiv* in this context. Fortunately, a single step of abstraction solves the problem. We abstract the body of the expression procedure,

$$\{y\neq 0\}\ idiv(x,y) \leftarrow idiv2(x,y)$$
$$idiv2(x,y) \leftarrow E(x,y).$$

Now if *idiv* is called in the context,

$$\ldots\ \{b\neq 0\}\ idiv(a,b)\ldots,$$

we can apply the expression procedure, replacing the call to the safe version of *idiv* by one directly to *idiv2*:

$$\ldots idiv2(a,b)\ldots.$$

Thus, we can do away with the expression procedure call and retain the improvement.

This technique allows commonly occurring subprograms to be coded in a very conservative fashion, since we can always eliminate extra safety checks by transformations.

In fact, in many of our previous examples, we were implicitly assuming that the variables were restricted to particular domains, such as nonnegative integers or lists. Qualifiers give us a way of making these domain restrictions explicit. Body qualifiers can then be used to demonstrate that the restrictions are met on recursive calls and also on calls to primitive functions meaningful only over a limited domain. (For example, we can use the qualifier mechanism to prove that $hd(\Lambda)$ can never be evaluated in the body of the *last* or *rev* functions.)

8. Notations

The use of *notations* provides us with a limited "control abstraction" mechanism for program derivations.

Like a primitive symbol, a notation is used in a program to stand for a set of equivalent methods for computing a value, without commitment to any particular one of them. Thus, notations provide both syntactic convenience and a mechanism for postponing commitment in program derivations. For example, Manna and Waldinger invented a notation for universal quantification over the elements of a list [Manna79]. Let $P(s)$ be a boolean valued term where s denotes an element of a domain S and P is a function symbol (in the terminology of Chapter 2). Then $P(\mathbf{all}(u))$ for a list u of elements of S is equivalent to the conjunction,

$$P(hd(u)) \wedge P(hd(tl(u))) \wedge P(hd(tl(tl(u)))) \wedge \ldots,$$

where there is one conjunct for every element of the list u. (By convention, we agree that $P(\mathbf{all}(\Lambda))$ is true.) The use of the **all** notation allows a programmer to postpone commitment to a particular order of testing the conjuncts until he has determined which order has the greatest potential for efficiency improvement.

Computational meaning can be given to notations by extending the language of programs to include a new type of function symbol, the *notation symbol*. Notation symbols are not given meaning externally, as are primitive symbols, and they are not defined by basic definitions, as are derivative symbols. Instead, they are defined only by expression procedures. Thus, the meaning of a term that includes a notation can be determined only after a suitable expression procedure has been applied.

We can use an expression procedure *schema* to describe a method for computing $P(\mathbf{all}(u))$, given a method for computing P:

$$P(\mathbf{all}(u)) \leftarrow \mathbf{if}\ u = \Lambda\ \mathbf{then\ true\ else}\ P(hd(u)) \wedge P(\mathbf{all}(tl(u)))$$

In the **else** clause of this expression procedure schema, the term $P(hd(u))$ is evaluated conventionally, while the recursive instance $P(\mathbf{all}(tl(u)))$ must be evaluated using the expression procedure, since this is the only definition involving the notation **all**.

(Use of such schemas requires a more sophisticated pattern matching mechanism in the evaluator. Our intent, however, is to use notations only in the *process* of program derivation, not in final programs. In either case, in order to give the schemas precise meaning it is necessary to do second order matching, as described in [Huet78].)

We illustrate these ideas by means of two examples. In the first example, we derive a program that tests whether all the elements of one list are less than all the elements of another list, in some partial order. A similar program is derived in [Manna79] using different techniques. (We compare the approach of Manna and Waldinger with ours in a later section.)

Allall. Using the all scheme, we can develop a program for testing whether all the elements in a list s are less than all the elements in a list t; that is, testing

$$\mathbf{all}(s) < \mathbf{all}(t),$$

for a given relation $<$. An instance of the expression procedure for **all** is

$$\mathbf{all}(s) < \mathbf{all}(t) \leftarrow \mathbf{if}\ u = \Lambda\ \mathbf{then\ true}\ \mathbf{else}\ [hd(s) < \mathbf{all}(t)] \wedge [\mathbf{all}(tl(s)) < \mathbf{all}(t)].$$

Another instance of the expression procedure for **all** is

$$x < \mathbf{all}(t) \leftarrow \mathbf{if}\ t = \Lambda\ \mathbf{then\ true\ else}\ [x < hd(t)] \wedge [x < \mathbf{all}(tl(t))].$$

If we have a way of computing the relation $x < y$ for list elements x and y, then these two expression procedures specify the desired program.

$$\begin{aligned}\mathbf{all}(s) < \mathbf{all}(t) &\leftarrow \mathbf{if}\ u = \Lambda\ \mathbf{then\ true}\\ &\qquad \mathbf{else}\ [hd(s) < \mathbf{all}(t)] \wedge [\mathbf{all}(tl(s)) < \mathbf{all}(t)]\\ x < \mathbf{all}(t) &\leftarrow \mathbf{if}\ t = \Lambda\ \mathbf{then\ true\ else}\ [x < hd(t)] \wedge [x < \mathbf{all}(tl(t))]\end{aligned}$$

In order to obtain an efficient implementation of this program, we need only to replace the complex names by simple ones using abstraction and application. We first abstract twice to obtain

$$\begin{aligned}\mathbf{all}(s) < \mathbf{all}(t) &\leftarrow aa1(s,t)\\ aa1(s,t) &\leftarrow \mathbf{if}\ u = \Lambda\ \mathbf{then\ true}\\ &\qquad \mathbf{else}\ [hd(s) < \mathbf{all}(t)] \wedge [\mathbf{all}(tl(s)) < \mathbf{all}(t)]\\ x < \mathbf{all}(t) &\leftarrow aa2(x,t)\\ aa2(x,t) &\leftarrow \mathbf{if}\ t = \Lambda\ \mathbf{then\ true\ else}\ [x < hd(t)] \wedge [x < \mathbf{all}(tl(t))].\end{aligned}$$

Three uses of the application rule yield

$$\begin{aligned}
&\mathbf{all}(s) < \mathbf{all}(t) \leftarrow aa1(s,t)\\
&\quad aa1(s,t) \leftarrow \mathbf{if}\ u = \Lambda\ \mathbf{then\ true\ else}\ aa2(hd(s),t) \wedge aa1(tl(s),t)\\
&\quad x < \mathbf{all}(t) \leftarrow aa2(x,t)\\
&\quad aa2(x,t) \leftarrow \mathbf{if}\ t = \Lambda\ \mathbf{then\ true\ else}\ [x < hd(t)] \wedge aa2(x,tl(t)).
\end{aligned}$$

Finally, we can eliminate the definition for $x < \mathbf{all}(t)$, since it is no longer referenced. The remaining three definitions comprise a program for the phrase $\mathbf{all}(s) < \mathbf{all}(t)$, where s and t are lists.

$$\begin{aligned}
&\mathbf{all}(s) < \mathbf{all}(t) \leftarrow aa1(s,t)\\
&\quad aa1(s,t) \leftarrow \mathbf{if}\ u = \Lambda\ \mathbf{then\ true\ else}\ aa2(hd(s),t) \wedge aa1(tl(s),t)\\
&\quad aa2(x,t) \leftarrow \mathbf{if}\ t = \Lambda\ \mathbf{then\ true\ else}\ [x < hd(t)] \wedge aa2(x,tl(t)).
\end{aligned}$$

(Observe that this program cannot be improved to a program that computes the maximum of s and the minimum of t and compares them, since the $<$ ordering may not be total.)

Ellipsis. A notation frequently used in mathematics is the *ellipsis* notation. By introducing appropriate definitions, we can make use of this notation in program derivations. For example, we can use the notation $P_1 \vee \cdots \vee P_n$ to stand for the disjunction

$$P_1 \vee P_2 \vee \cdots \vee P_n.$$

(The ellipsis in the latter disjunction is a genuine ellipsis; the ellipsis in the first conjunction is a notation.)

Besides being a syntactic convenience, the ellipsis notation enables a programmer to postpone commitment to an order of computation of the conjunction. He could, for example, use any one of the following set of equivalent definitions for the computation:

$$\begin{aligned}
P_i \vee \cdots \vee P_j \leftarrow\ &\mathbf{if}\ i = j\ \mathbf{then}\ P_i\\
&\mathbf{else}\ i < k < j \wedge \big((P_i \vee \cdots \vee P_{k-1}) \vee (P_k \vee \cdots \vee P_j)\big),
\end{aligned}$$

(where k is an abbreviation for a function of i and j). Alternative definitions are

$$\begin{aligned}
P_i \vee \cdots \vee P_j \leftarrow\ &\mathbf{if}\ i = j\ \mathbf{then}\ P_i\\
&\mathbf{else}\ P_i \vee (P_{i+1} \vee \cdots \vee P_j),
\end{aligned}$$

and

$$\begin{aligned}
P_i \vee \cdots \vee P_j \leftarrow\ &\mathbf{if}\ i = j\ \mathbf{then}\ P_i\\
&\mathbf{else}\ (P_i \vee \cdots \vee P_{j-1}) \vee P_j.
\end{aligned}$$

The above definitions are all implicitly qualified by $(i \leq j)$. We could also admit empty disjunctions, as in

$$\begin{aligned}
P_i \vee \cdots \vee P_j \leftarrow\ &\mathbf{if}\ i > j\ \mathbf{then\ false}\\
&\mathbf{else}\ P_i \vee (P_{i+1} \vee \cdots \vee P_j).
\end{aligned}$$

Here, no qualifier is needed.

A membership test program could thus be specified using this notation:

$$\underline{\text{Compute}}\colon\ A[1] = key \vee \cdots \vee A[M] = key.$$

The first schema could be used to develop a binary search program; the others yield linear searches.

9. Proving Properties of Programs

In complicated program derivations, it is often necessary to use induction to prove a property of a program before a particular transformation, such as a simplification or the application of a qualified expression procedure can be applied. In this section, we informally describe the well-known method of *computation induction* as applied to our language of programs. Computation induction is described in detail in [Manna74] and in [Manna73].

The notation we use for assertions is the body qualifier notation introduced in an earlier section of this chapter. We will write a definition $E \leftarrow F$

as

$$E \ \{P\} \leftarrow F$$

to indicate that the *assertion* P holds whenever evaluation of E terminates. Thus, P is an "output condition" for E. For example, we may write

$$\{list(s) \wedge list(t)\}\ s \circ t\ \{list(s \circ t)\} \leftarrow \textbf{if } s = \Lambda \textbf{ then } t \textbf{ else } cons(hd(s), tl(s) \circ t)$$

to indicate that if the arguments to *append* are lists, then so will be the results. That is, if *append* terminates, then

$$list(s) \wedge list(t) \supset list(s \circ t).$$

We will often use a special notation for the results of functions, writing **result** to indicate a result value. Thus, in the example above we could have written

$$\{list(s) \wedge list(t)\}\ s \circ t\ \{list(\textbf{result})\} \leftarrow \textbf{if } s = \Lambda \textbf{ then } t \textbf{ else } cons(hd(s), tl(s) \circ t).$$

In program derivations assertions are used primarily as a means of making simplifications. To introduce an assertion about a function into an expression containing an instance of that function name, we use the ***tag promotion*** rule: Consider the qualified definition,

$$\{Q\}\ E\ \{P\} \leftarrow F.$$

This means that whenever Q is true, we can evaluate E using this definition, and further that if evaluation of E terminates, P will be true. Thus, if evaluation of E terminates, then $Q \supset P$. Therefore, we can add the body tag P at any occurrence of E for which Q is true and whose evaluation terminates.

Computation Induction. *Computation induction* is the principal method we use for proving properties of recursively defined derivative names. Suppose we want to prove a property $P(\bar{x}, f(\bar{x})) \equiv P(\bar{x}, \textbf{result})$ holds whenever evaluation of instances of a basic name $f(\bar{x})$ terminates. Let f have the basic definition $f(\bar{x}) \leftarrow E$. By computation induction, it is sufficient to show that the body qualifier $\{P(\bar{x}, \textbf{result})\}$ holds for the body E when we assume the induction hypothesis $P(\bar{u}, f(\bar{u}))$ holds at each instance $f(\bar{u})$ in definition bodies in the program.

Recall that all derivative names are defined by basic definitions; therefore we can inductively prove properties of a derivative name using only the appropriate basic definitions, and we will not need to extend the induction rule to the case of expression procedure definitions.

Suppose, for example, that our program consists of the definition

$$f(x) \leftarrow \ldots f(u) \ldots,$$

which contains a recursive call $f(u)$, where u is a term. To prove that $P(x, f(x))$ holds whenever evaluation of $f(x)$ terminates, we assume the induction hypothesis for the recursive call to f.

$$f(x) \leftarrow \ldots \ \{P(u, f(u))\}\ f(u) \ldots$$

We now attempt to prove (possibly using our transformation techniques) that the body qualifier $\{P(x, \textbf{result})\}$ can be introduced in the body of f:

$$f(x) \leftarrow \ \{P(x, \textbf{result})\} \ldots f(u) \ldots$$

If so, then the induction is successful and we can add the assertion to the definition of f.

$$f(x)\ \{P(x, \textbf{result})\} \leftarrow \ldots f(u) \ldots$$

Fibonacci. We illustrate this technique with a very simple example: We prove that the result of the Fibonacci function,

$$\{x \geq 0\}\ fib(x) \leftarrow \textbf{if } x \leq 1 \textbf{ then } x \textbf{ else } fib(x-1) + fib(x-2),$$

is nonnegative whenever the parameter x is. To show this, we introduce appropriate induction hypotheses,

$$\{x \geq 0\}\ fib(x) \leftarrow \textbf{if } x \leq 1 \textbf{ then } x \textbf{ else } \{fib(x-1) \geq 0\}\ \{fib(x-2) \geq 0\}\ fib(x-1) + fib(x-2)$$

and attempt to establish $\{\textbf{result} \geq 0\}$ for the definition body.

Since x is nonnegative, we can introduce a body qualifier in the **then** clause,

$$\{x \geq 0\}\ fib(x) \leftarrow \textbf{if } x \leq 1 \textbf{ then } \{x \geq 0\}\ x \textbf{ else } \{fib(x-1) \geq 0\}\ \{fib(x-2) \geq 0\}\ fib(x-1) + fib(x-2),$$

and hence we have the qualifier $\{fib(x-1)+fib(x-2) \geq 0\}$ over the **else** clause. Therefore, using the body qualifiers rules presented earlier, we can introduce the full body qualifier $\{(\textbf{if } x \leq 1 \textbf{ then } x \textbf{ else } fib(x-1)+fib(x-2)) \geq 0\}$ or more simply $\{result \geq 0\}$. Thus, our induction is successful, and we can add the assertion to the definition of *fib*.

$$\{x \geq 0\}\ fib(x)\ \{result \geq 0\} \leftarrow \textbf{if } x \leq 1 \textbf{ then } x \textbf{ else } fib(x-1) + fib(x-2)$$

This assertion can now be promoted into any context in which *fib* appears.

Append. We illustrate the use of the computation induction rule in transformation with an example involving the *append* function introduced earlier:

$$s \circ t \leftarrow \textbf{if } s = \Lambda \textbf{ then } t \textbf{ else } cons(hd(s), tl(s) \circ t).$$

Observe that if s and t are lists, then evaluation of *append* always terminates. We saw in a previous example that an expression procedure for $\Lambda \circ t$ can easily be derived using the composition rule. It is more difficult, however, to derive the analagous definition for $s \circ \Lambda$. Consider the following specialized version of *append*, obtained by composition:

$$s \circ \Lambda \leftarrow \textbf{if } s = \Lambda \textbf{ then } \Lambda \textbf{ else } cons(hd(s), tl(s) \circ \Lambda).$$

We are not able to transform this to the equivalent definition,

$$s \circ \Lambda \leftarrow s,$$

using only abstraction, composition, application, and simplification.

With the computation induction rule, however, this transformation can easily be achieved. Our induction hypothesis is simply $\{result = s\}$:

$$s \circ \Lambda \leftarrow \textbf{if } s = \Lambda \textbf{ then } \Lambda \textbf{ else } cons(hd(s), \{tl(s) \circ \Lambda = tl(s)\}\ tl(s) \circ \Lambda).$$

But if $tl(s) \circ \Lambda = tl(s)$ then

$$cons(hd(s), tl(s) \circ \Lambda) = cons(hd(s), tl(s)) = s,$$

so we have

$$s \circ \Lambda \leftarrow \textbf{if } s = \Lambda \textbf{ then } \{\Lambda = s\}\ \Lambda \textbf{ else } \{cons(hd(s), tl(s) \circ \Lambda) = s\}\ cons(hd(s), tl(s) \circ \Lambda),$$

or

$$s \circ \Lambda \leftarrow \{(\textbf{if } s = \Lambda \textbf{ then } \Lambda \textbf{ else } cons(hd(s), tl(s) \circ \Lambda)) = s\}\ \textbf{if } s = \Lambda \textbf{ then } \Lambda \textbf{ else } cons(hd(s), tl(s) \circ \Lambda),$$

or

$$s \circ \Lambda \leftarrow \{result = s\}\ \textbf{if } s = \Lambda \textbf{ then } \Lambda \textbf{ else } cons(hd(s), tl(s) \circ \Lambda),$$

so the induction is successful. Since evaluation of $s \circ \Lambda$ always terminates we can use the assertion to replace the body by s (by the Program Substitution Theorem), obtaining the new definition

$$s \circ \Lambda \leftarrow s.$$

10. Semantic Expression Procedures

The composition rule provides a purely syntactic way of introducing expression procedures. Often there are facts that we would like to introduce as expression procedures, but which require for their proof reasoning beyond the power of this simple rule system. In these instances, it may be more appropriate to use a semantically based method for introducing the new definitions. We describe here a technique called ***semantic composition*** for doing this. (In this section we make use of the terminology and results of Chapter 2.)

The results of Chapter 2 imply that if we establish a potential expression procedure to be a progressive pair, then we can add that expression procedure to a uniform program, and the resulting program will be uniform as well, and so we can correctly apply the transformation rules to it. We illustrate this method with two examples.

Exponential. Consider the following exponentiation program over the nonnegative integers:

$$x^n \leftarrow \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } x \cdot x^{n-1}$$

We can prove that this program terminates using a simple measure ordering K_P constructed according to the method described in Section 2.8. We simply define the function M mapping ground basic terms to multisets over the well-founded set of nonnegative integers,

$$M[\![x^n]\!] = \{n\},$$

(and assume that all other functions are primitive). Thus,

$$K_P[\![\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } x \cdot x^{n-1}]\!] = \begin{cases} \{n-1\}, & \text{if } n > 0 \\ \emptyset, & \text{if } n = 0, \end{cases}$$

which implies that

$$K_P[\![x^n]\!] \gg K_P[\![\textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } x \cdot x^{n-1}]\!],$$

for all n, and so evaluation of x^n terminates for all n in the domain.

Using this underlying ordering, we can easily prove that the conditional definition

$$\{a>1 \wedge b>1\}\ x^{a \cdot b} \leftarrow (x^a)^b$$

is a progressive pair: Consistency can be proved by ordinary computation induction on the basic definition of x^n. Progressiveness follows from the fact that

$$a > 1 \ \wedge \ b > 1 \quad \supset \quad \{a \cdot b\} \gg \{a, b\},$$

which implies that

$$K_P[\![x^{a \cdot b}]\!] \gg K_P[\![(x^a)^b]\!].$$

This implies that the program

$$x^n \leftarrow \textbf{if } n = 0 \textbf{ then } 1 \textbf{ else } x \cdot x^{n-1}$$
$$\{a>1 \wedge b>1\}\ x^{a \cdot b} \leftarrow (x^a)^b$$

is uniform, and so we can correctly apply the transformation rules to this program. Use of the expression procedure enables us to compute $x^{a \cdot b}$ in only $a + b$ iterations, rather than in $a \cdot b$ iterations, as would be required with the conventional method. We could then use the transformation rules to derive a new exponentiation program that would compute x^n in only $\log(n)$ time.

These techniques can also be used in cases where the initial program is not basic. In these cases, we must find a measure ordering such that all expression procedures are progressive pairs. (In some examples, it may be necessary to introduce a fairly complicated ordering in order to do this. Some orderings that may be useful are discussed in [Dershowitz79a] and [Dershowitz79b].)

Reverse. As another example, consider again the *rev* function over lists,

$$rev(x) \leftarrow \textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(x)) \circ cons(hd(x), \Lambda).$$

It may be useful in some derivations to have an expression procedure for $rev(a \circ b)$, such as

$$rev(a \circ b) \leftarrow rev(b) \circ rev(a).$$

In order to introduce this expression procedure using semantic composition, we must show that it is consistent and progressive. Consistency can be proved by a straightforward inductive proof on a. To prove that it is progressive, we use the measure ordering induced by

$$M[\![rev(a)]\!] = |a|,$$

where $|a|$ is the length of the list a. Now $|a \circ b| = |a| + |b|$, so

$$a \neq \Lambda \wedge b \neq \Lambda \quad \supset \quad \{|a \circ b|\} \gg \{|a|, |b|\},$$

which implies that the expression procedure is progressive when neither a nor b are empty lists.

$$\{a \neq \Lambda \wedge b \neq \Lambda\}\ rev(a \circ b) \leftarrow rev(b) \circ rev(a).$$

We could also introduce the unqualified expression procedure,

$$rev(a \circ cons(b, \Lambda)) \leftarrow cons(b, rev(a)),$$

using the same ordering. Observe that $|cons(b, \Lambda)| = 1$, so for any list a,

$$\{|a \circ cons(b, \Lambda)_1|\} \gg \{|a|\},$$

which implies that this expression procedure is progressive.

An example derivation. We illustrate the use of the above expression procedure in a derivation of a program for computing $rev(tl(rev(x)))$ for a nonempty list x.

In this derivation we will need an expression procedure for computing $tl(s \circ t)$ where s and t are lists: Composing tl with *append* yields

$$tl(s \circ t) \leftarrow tl\big(\textbf{if } s = \Lambda \textbf{ then } t \textbf{ else } cons(hd(s), tl(s) \circ t)\big),$$

which simplifies to

$$tl(s \circ t) \leftarrow \textbf{if } s = \Lambda \textbf{ then } tl(t) \textbf{ else } tl\big(cons(hd(s), tl(s) \circ t)\big),$$

or

$$tl(s \circ t) \leftarrow \textbf{if } s = \Lambda \textbf{ then } tl(t) \textbf{ else } tl(s) \circ t.$$

The derivation process for $rev(tl(rev(x)))$ consists of a single step of specialization. We start by composing the definition of rev with the inside instance of rev in the expression $rev(tl(rev(x)))$:

$$rev(tl(rev(x))) \leftarrow rev\big(tl\big(\textbf{if } x = \Lambda \textbf{ then } \Lambda \textbf{ else } rev(tl(x)) \circ cons(hd(x), \Lambda)\big)\big).$$

Since we are only interested in the case where $x \neq \Lambda$, we compose again to add an appropriate qualifier. This implies that the **if** test is always false, so we eliminate the **then** clause to get

$$\{x \neq \Lambda\}\ rev(tl(rev(x))) \leftarrow rev\big(tl\big(rev(tl(x)) \circ cons(hd(x), \Lambda)\big)\big).$$

The expression procedure we derived above for $tl(s \circ t)$ can now be applied, giving

$$\{x \neq \Lambda\}\ rev(tl(rev(x))) \leftarrow rev\big(\textbf{if } rev(tl(x)) = \Lambda \textbf{ then } tl(cons(hd(x), \Lambda)) \textbf{ else } tl(rev(tl(x))) \circ cons(hd(x), \Lambda)\big).$$

Distributing the outer rev call over the conditional yields

$$\{x \neq \Lambda\}\ rev(tl(rev(x))) \leftarrow \textbf{if } rev(tl(x)) = \Lambda \textbf{ then } rev(tl(cons(hd(x), \Lambda))) \textbf{ else } rev\big(tl(rev(tl(x))) \circ cons(hd(x), \Lambda)\big).$$

We now simplify the conditional clause by clause. To simplify the **if** clause, we apply the expression procedure,

$$(rev(x) = \Lambda) \leftarrow (x = \Lambda),$$

that we derived in an earlier section.

$$\{x \neq \Lambda\}\ rev(tl(rev(x))) \leftarrow \textbf{if } tl(x) = \Lambda \textbf{ then } rev(tl(cons(hd(x), \Lambda))) \textbf{ else } rev\big(tl(rev(tl(x))) \circ cons(hd(x), \Lambda)\big)$$

To simplify the **then** clause, we first use properties of tl and $cons$,

$$\{x \neq \Lambda\}\ rev(tl(rev(x))) \leftarrow \textbf{if } tl(x) = \Lambda \textbf{ then } rev(\Lambda) \textbf{ else } rev\big(tl(rev(tl(x))) \circ cons(hd(x), \Lambda)\big),$$

and then apply the definition of rev and simplify the result:

$$\{x \neq \Lambda\}\ rev(tl(rev(x))) \leftarrow \textbf{if } tl(x) = \Lambda \textbf{ then } \Lambda \textbf{ else } rev\big(tl(rev(tl(x))) \circ cons(hd(x), \Lambda)\big).$$

Finally, to simplify the **else** clause, we apply the second semantic expression procedure we developed above for $tl(s \circ t)$ to obtain the definition,

$$\{x \neq \Lambda\}\ rev(tl(rev(x))) \leftarrow \textbf{if } tl(x) = \Lambda \textbf{ then } \Lambda \textbf{ else } cons(hd(x), rev(tl(rev(tl(x))))),$$

which is recursive.

Thus, our specialization is successful, and we need only to rename. Abstraction and application yield the final pair of definitions,

$$\{x \neq \Lambda\}\ rev(tl(rev(x))) \leftarrow rtr(x)$$
$$rtr(x) \leftarrow \textbf{if } tl(x) = \Lambda \textbf{ then } \Lambda \textbf{ else } cons(hd(x), rtr(tl(x))).$$

The function $rtr(x)$ produces a list identical to x except the last element is removed.

Chapter 4

Comparison with other Systems

In this chapter we compare our approach to program derivation with several other approaches. We first conduct a brief survey of related work; we then detail the relation of our system to the systems of Burstall and Darlington and of Manna and Waldinger.

1. Survey of Related Work

Much of the early research in program transformation, dealing with the improvement of programs by compilers, led to the development of what is now a large and useful set of compile-time optimization transformations, such as common-subexpression-elimination and code-motion-out-of-loops. (See [Aho-73], [Cocke71], [Cocke77], and [Earnest74].) This work naturally encouraged exploration of the use of program transformation techniques in the programming process itself. We consider here the goals of such an exploration, the techniques that have been developed, and the areas in which these techniques have been successfully applied.

Goals There is a widespread feeling that a transformational programming discipline will improve our ability to produce correct and understandable programs. This transformational approach to programming methodology has been advocated by researchers in automatic program synthesis ([Bauer78], [Burstall77], [Manna79]), by developers of programming tools ([Cheatham72], [Bates79]), and others ([Aiello78], [Clark77], [Dijkstra75], [Knuth74b], [McKeeman75]), and is consistent with the classical presentations of structured programming and stepwise refinement, [Dahl72], [Dijkstra76], [Dijkstra77], and [Wirth71]. (See also the discussion of this approach in Chapter 6.)

Techniques. A variety of transformational techniques have been developed for the improvement of iterative programs — those programs with imperative features such as assignment and destructive modification of data structures. Many of these methods are natural outgrowths of compiler optimization techniques. (See [Arsac79], [Gerhart75], [Loveman77], [Paige79], [Standish76a], and [Standish76b].) Some of this work has concentrated on the transformation of recursive programs into more easily implemented iterative programs. (See [Strong71], [Steele76], [Bird77a], [Huet78], [Darlington74], and [Darlington72].)

More recently, there has been considerable interest in manipulating applicative programs -- programs with no imperative features such as assignment. This lack of imperative features allows recursive definitions in programs to be treated very much like equations; thus, the transformation rules in such systems tend to take on an elegant mathematical character. (See [Burstall77], [Manna79], [Manna80], and [Wegbreit76].)

The seeds for much of this research on the transformation of recursive programs are found in the book of Rosza Péter [Péter51]. Besides presenting a vast selection of examples of recursive programs, she develops a variety of techniques for transforming them.

The program transformation system presented here, consisting of the four rules, abstraction, application, composition, and definition elimination, is quite similar in spirit to the transformation system developed and implemented by Burstall and Darlington in Edinburgh [Burstall77], and to the DEDALUS system of Manna and Waldinger [Manna79]. Both of these systems are based on essentially the same set of transformation rules, which we call the *UF rules* (Unfold-Fold rules). We describe these rules later in this chapter.

Burstall and Darlington observed that this system of transformation rules has the drawback that termination of programs is not always preserved. There have been several approaches to this problem. In the system of Manna and Waldinger, equivalence is guaranteed through explicit, automatically generated termination proofs; this approach is only useful when the initial program terminates, however. (Thus, we could not derive, say, the top-down parsing algorithm using this method.) Kott, on the other hand, showed how the sequence of applications of the *fold* and *unfold* transformation rules can be restricted in various ways to ensure strong equivalence is preserved, although

it is not yet clear how his restrictions may be applied to the development of automatic systems [Kott78].

The emphasis in Burstall and Darlington's original work is on the statement and use of the transformation rules. Manna and Waldinger concentrate more on the heuristic aspect of the use of the rules; they develop a versatile automatic implementation based on backwards inference and subgoaling. In addition, they consider problems related to the development of programs with assignments and other imperative features [Manna75], [Manna79]. Other investigations into the heuristic aspect of the UF rules include [Darlington78a], [Feather79], and [Peterossi77].

Our approach, using expression procedures, has been to develop transformation rules that preserve strong equivalence of programs, eliminating the need for termination proofs during transformation and for restrictions on the sequence of application of transformation rules. The expression procedure approach also provides a *natural framework for the technique of specialization*, which, as our examples show, underlies much of the activity of program transformation. Indeed, many of the standard derivation examples developed using the UF system ([Burstall77], [Manna79]) reduce in our framework to a single step of specialization.

In fact, we are able to develop all of the examples of [Burstall77] in our framework, though one of them (*frontier*) requires a slightly extended composition rule.

Many recursive programs are inefficient simply because of the high degree of redundancy in their recursive calls. While we can often transform such programs to eliminate this redundancy, there are examples that are easily treated by general techniques such as dynamic programming. In the parser derivations of Chapter 5, for example, after obtaining recursive versions of the desired algorithms by a series of specialization steps, we use a dynamic programming technique to eliminate the redundancy among the recursive calls. (See the Appendix for an outline of this technique.) Dynamic programming is described in [Aho74], a variety of transformations based on it are described in [Cohen79], [Bird77b], and [Sickel79].

Applications. Besides influencing our methodology of programming, these techniques have direct applications in several areas of research. We

consider here applications to the development of programming tools, such as program synthesis systems and very high level languages, and to the exposition of complicated algorithms.

The development of systems for the interactive or automatic synthesis of programs will very likely have a significant impact on programming productivity. The UF system has been implemented both by Burstall and Darlington and by Manna and Waldinger, to allow experimentation with strategies for applying the transformation rules. (We discuss these two systems below.) Feather has shown how the UF system can be applied to large program development problems [Feather79].

There are several other approaches to the development of automatic program synthesis systems based on program transformation techniques. (See e.g., [Barstow77].)

Another important area of application for program transformation techniques is in the implementation of very high level languages such as SETL. (See [Schwartz73] and [Schonberg79].) Of particular interest is the work of Paige [Paige77], in which he describes a program improvement technique for SETL programs that is similar in function to our specialization technique. His approach, called *formal differentiation* of programs, is based on the reduction-in-strength optimization, and is an extension of earlier work by Earley [Earley74].

Transformation techniques have also been successfully exploited to describe complex algorithms. An example of this approach to the explanation of algorithms appears in [Knuth74a]. Tarjan's historical survey of the improvement in complexity results of combinatorial algorithms shows how several seemingly unrelated algorithms were all improved using a relatively small set of high level transformations, such as path compression and depth-first graph search [Tarjan77]. Developmental taxonomies of sorting algorithms have been developed independently at Edinburgh, using the UF system, and at Stanford, using rule-based techniques. (See [Darlington78a], [Green77], [Barstow80], and [Clark80].) Other examples of transformational derivations are in [Correll78], [Dewar80], and in Chapter 5 of this work.

Outline of this chapter. In this chapter, we examine the UF system in some detail. We compare the use of this system by Burstall and Darlington,

and by Manna and Waldinger, by means of a simple example. We also discuss issues related to the correctness of the UF transformation rules.

2. UF-Programs and their Evaluator

We state the UF transformation rules in the context of a language of programs that is essentially our language of *basic* programs extended to allow multiple definitions of derivative symbols. (This material is presented using the terminology of Chapter 2.)

A *UF-program* is an expression of the form

$$f_1, \ldots, f_m : \ d_1; \ldots; d_n \ ,$$

where the f_i are derivative symbols and the d_j are basic definitions. As before, the f_i are called the *principal* symbols of the program.

A UF-program is *well-formed* if

(a) all of its definitions are well-formed, and

(b) every derivative name that occurs in the name or body of a definition or is principal is defined by *at least one* basic definition.

The evaluator for UF-programs, *EvUF*, is essentially the same as the call-by-value basic program evaluator *Evb*, except a nondeterministic choice is made when more than one definition is applicable.

Given a UF-program B and a set of primitive simplification rules S, the procedure *EvUF*(t) on legal input terms for B is defined as follows:

EvUF(t):

(1) (*Simplify*) Let t be $simp[t]$.

(2) If t is now primitive, then exit the evaluator with t as the result.

(3) (*Choose*) Since B is well formed, there is at least one definition $f(\bar{x}) \leftarrow s$ such that there is a term t',

$$t = t'[f(\bar{u})] \ \textit{for}\ z,$$

where each of the terms u_i is primitive and z occurs exactly once in t', and in a leftmost position. (f is a ***leftmost-innermost*** choice.)

(4) (*Expand*) Let t be $t'[s[\bar{u}] \ \textit{for}\ \bar{x}] \ \textit{for}\ z$.

(5) Go to step (1). ∎

The call-by-name evaluator *EvUFn* is identical to *EvUF*, except that a leftmost-outermost choice is used in Step 3 (see Chapter 2).

Since this evaluator is nondeterministic, it is necessary to define a notion of uniformity. A UF-program is *uniform* if all terminating evaluations of *EvUF* have the same result. Thus two programs are *equivalent* if they are both uniform and if one has a terminating evaluation then so does the other, with the same result. A program is *fully uniform* if it is uniform and *in addition* has the universal termination property (i.e., either all evaluation paths terminate or none of them do).

3. The UF Transformations

Besides simplification rules, there are four transformation rules in the UF system. The first three rules add definitions to a program, but do not otherwise modify the program. The last transformation serves only to remove definitions from a program. We state the call-by-value versions of the rules.

Application (Unfolding). Let $s \leftarrow u$ be a definition in a given program P, and (s', u') be a proper definition instance. Suppose P has a definition of the form

$$t \leftarrow t'[s'] \ \textit{for}\ z,$$

where the variable z occurs exactly once in the term t'. The new program P' is obtained by adding to P the new definition

$$t \leftarrow t'[u'] \ \textit{for}\ z.$$

Folding. Let $s \leftarrow u$ be a definition in a given program P, and (s', u') be a proper definition instance. Suppose P has a definition of the form

$$t \leftarrow t'[u'] \ \textit{for}\ z,$$

where the variable z occurs exactly once in the term t'. The new program P' is obtained by adding to P the new definition

$$t \leftarrow t'[s'] \text{ for } z.$$

Definition Introduction. Let P be a given program. Let

$$f(x_1, \ldots, x_n) \leftarrow t$$

be a well-formed basic definition such that the derivative name f does not occur anywhere in the program P. Then a new program P' is obtained by adding this new definition to P.

Definition Elimination. (a) Let P be a given program with two or more definitions of a derivative name f. A new program P' is obtained by removing any one of these definitions. (b) Let P be a given program with one or more definitions of a derivative name f such that f is not principal and does not occur in any other definitions in P. A new program P' is obtained by dropping these definitions of f from P.

These rules are stated for the call-by-value evaluator *EvUF*. For the call-by-name evaluator *EvUFn*, the rules are identical, except the definition instances used in the folding and unfolding rules need only be ***call-by-name instances***: Recall that a call-by-name instance of a definition $f(\bar{x}) \leftarrow t$ is any pair $\langle f(\bar{u}), t[\bar{u}] \rangle$ *for* $\bar{x}$ where $\bar{u}$ are any terms.

4. The Correctness of the UF Transformations

The basic result concerning the UF transformation rules is summarized in the following theorem.

Theorem: Let a UF-program P_2 be obtained from a UF-program P_1 by a sequence of applications of UF transformations. Then:

(a) If P_1 is uniform, then so is P_2.

(b) If there is a terminating evaluation path for P_2, then there is one for P_1, and with the same result.

Proof. (Outline) This is essentially an extended version of the Program Substitution Theorem (see Chapter 2), and can be proved using similar techniques. Observe that while the first three rules add new definitions to a program and thus new evaluation paths, they never add a terminating path where there was none previously. It is possible, however, that they can add nonterminating paths where there were none previously. ∎

Burstall and Darlington state and give an informal proof of a version of the Program Substitution Theorem in [Burstall77]. A similar theorem is stated using an algebraic formulation in [Kott78].

It is straightforward to strengthen this result to show that if P_1 is fully uniform, and if only the the application, definition introduction, and definition elimination transformations are used, then P_2 will be fully uniform, and equivalent to P_1.

If the fold transformation is used, however, then full uniformity or even equivalence may not always be preserved. Consider the fully uniform program

$$f: \quad f(x) \leftarrow x.$$

If this definition is folded with *itself*, then the resulting program,

$$f: \qquad f(x) \leftarrow x\,; \qquad f(x) \leftarrow f(x),$$

is not fully uniform. Now, if the first definition is eliminated by the definition elimination rule, then the resulting program is the always looping program,

$$f: \quad f(x) \leftarrow f(x),$$

which, by our definition above, is not equivalent to the initial program.

It is not easy to restrict the use of the UF transformations to avoid this pitfall. Kott presents a solution to this problem in which the sequence of UF transformations is restricted in various ways to ensure equivalence [Kott78]. We discuss this below.

5. An Example

The use of the UF system as contrasted to ours is best illustrated by means of a simple example. In this example (taken from [Burstall77]), two parallel recursions are transformed into a single recursion.

UF System. We start with a simple recursive program for calculating a vector product:

$$p(a,b,n) \leftarrow \textbf{if } n=0 \textbf{ then } 0 \textbf{ else } a_n b_n + p(a,b,n-1).$$

Here, $p(a,b,n)$ stands for the product of the vectors a and b, which are both assumed to have length n. Our goal is to compute efficiently the value of expressions of the form

$$p(a,b,n) + p(c,d,n),$$

for some a, b, c, d, and n.

The first transformation step is to use the definition introduction transformation to introduce a definition corresponding to this specification:

$$f(a,b,c,d,n) \leftarrow p(a,b,n) + p(c,d,n).$$

The two instances of p in this definition body are now both unfolded to obtain the new definition

$$f(a,b,c,d,n) \leftarrow \big(\textbf{if } n=0 \textbf{ then } 0 \textbf{ else } a_n b_n + p(a,b,n-1)\big) + \big(\textbf{if } n=0 \textbf{ then } 0 \textbf{ else } c_n d_n + p(c,d,n-1)\big),$$

which simplifies to

$$f(a,b,c,d,n) \leftarrow \textbf{if } n=0 \textbf{ then } 0 \textbf{ else } a_n b_n + c_n d_n + p(a,b,n-1) + p(c,d,n-1).$$

We observe that an instance of the initial computation specification occurs in the body of this definition, so it is natural to form a recursion. We use the fold rule to accomplish this, folding the first definition of f with the last:

$$f(a,b,c,d,n) \leftarrow \textbf{if } n=0 \textbf{ then } 0 \textbf{ else } a_n b_n + c_n d_n + f(a,b,c,d,n-1)$$

Observe that we needed to use two definitions of f *simultaneously* in order to introduce the recursion. (In fact, we conjecture that if multiple definitions are not allowed in the language of programs, then it is impossible to introduce nontrivial recursions using the UF rules.)

The final step of this derivation is the elimination of all the definitions of f except the first and the last:

$$f(a,b,c,d,n) \leftarrow p(a,b,n) + p(c,d,n)$$
$$f(a,b,c,d,n) \leftarrow \textbf{if } n=0 \textbf{ then } 0 \textbf{ else } a_n b_n + c_n d_n + f(a,b,c,d,n-1).$$

(The first of these definitions is required for introducing calls to f with the fold rule.)

Expression Procedure System. In our system, we obtain the same improvement through the use of an expression procedure and a single step of specialization. Starting with the same initial program and computation specification,

Compute: $p(a,b,n) + p(c,d,n)$
Given: $p(a,b,n) \leftarrow \textbf{if } n=0 \textbf{ then } 0 \textbf{ else } a_n b_n + p(a,b,n-1),$

we use the composition rule to obtain an expression procedure directly corresponding to the Compute specification:

$$p(a,b,n) + p(c,d,n) \leftarrow p(a,b,n) + \big(\textbf{if } n=0 \textbf{ then } 0 \textbf{ else } c_n d_n + p(c,d,n-1)\big)$$

Application is then used to expand the first instance of p to obtain

$$p(a,b,n) + p(c,d,n) \leftarrow \big(\textbf{if } n=0 \textbf{ then } 0 \textbf{ else } a_n b_n + p(a,b,n-1)\big) + \big(\textbf{if } n=0 \textbf{ then } 0 \textbf{ else } c_n d_n + p(c,d,n-1)\big),$$

which simplifies to

$$p(a,b,n) + p(c,d,n) \leftarrow \textbf{if } n=0 \textbf{ then } 0 \textbf{ else } a_n b_n + c_n d_n + p(a,b,n-1) + p(c,d,n-1),$$

which is recursive. Thus, our specialization is successful and we need only to rename. We abstract the body of this definition into a new function, which has

five parameters corresponding to the free variables in the expression procedure body.

$$p(a,b,n) \dotplus p(c,d,n) \leftarrow f(a,b,c,d,n)$$
$$f(a,b,c,d,n) \leftarrow \textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } a_n b_n + c_n d_n + p(a,b,n-1) + p(c,d,n-1).$$

Two uses of the application rule and elimination of the expression procedure then yield the final specialized program:

Compute: $f(a,b,c,d,n)$
Given: $f(a,b,c,d,n) \leftarrow \textbf{if } n = 0 \textbf{ then } 0 \textbf{ else } a_n b_n + c_n d_n + f(a,b,c,d,n-1).$

6. A Note on the Restrictions of Kott

Kott's restrictions to the UF system involve both the form of programs and the sequence of transformation rules that may be applied. We sketch his restrictions here: First, all programs considered are basic programs (i.e., each derivative name is defined exactly once). Second, a transformation sequence from a program P to a program P' must consist of a series of ***basic steps***, each of which modifies a single definition in the program. A step consists of a series of unfoldings, followed by a simplification step, followed by a series of foldings.

If the number of unfoldings exceeds the number of foldings in each step, and if certain conditions are met by the simplification operations, then P is strongly equivalent to P'. In certain cases, this restriction may be weakened, to require only that the number of unfoldings be greater than or equal to the number of foldings.

There are significant restrictions on the form of the simplification step.

For example, given the program P_1,

$$f(\bar{x}) \leftarrow E$$
$$h(\bar{z}) \leftarrow F,$$

a basic step for f starts with a number of unfoldings from the initial term $f(\bar{x})$. The first unfolding step clearly must unfold the definition of f itself, yielding E. Further unfoldings could involve either of the two definitions. A simplification step then follows this series of unfoldings. Finally, there can be a sequence of foldings, again using the two initial definitions. The result of this sequence of rule applications is a term H, which we substitute in the original definition of f to obtain the new program P_2,

$$f(\bar{x}) \leftarrow H$$
$$h(\bar{z}) \leftarrow F.$$

Thus, the sample UF derivation in the last section is just a single basic step, consisting of three unfoldings followed by a single folding. Observe that Kott obtains the effect of multiple definitions by postponing modification of the initial set of definitions until the end of the basic step.

What is the practical impact of these results? Kott's work is directed at settling an important theoretical question regarding the UF system, and so it does not address heuristic issues. In particular, while it may be that existing UF derivations can be modified to fit the restricted form, it is not yet clear how suitable this form is for use in an automatic system or for the formal exposition of algorithms.

7. UF Simulation of Expression Procedures

We note that the UF transformations can be used to simulate the introduction and application of expression procedures. An expression procedure can be represented in a UF-program by a *pair* of definitions. For example, the expression procedure $s \leftarrow u$ would be represented by the pair of definitions $f(\bar{x}) \leftarrow s$ and $f(z) \leftarrow u$ where $\bar{x} = vars[\![s]\!]$.

The application, abstraction, and composition rules can be simulated in a straightforward manner using this representation. For example, given a representation of the definition $s \leftarrow u$, an expression procedure application rule would transform a representation of the definition $r \leftarrow t[s[\bar{u}]]$ into a representation of the definition $r \leftarrow t[u[\bar{u}]]$. We illustrate this here:

The two initial definitions in this example are represented by the four definitions,

$$f(z) \Leftarrow s \qquad h(z) \leftarrow r$$
$$f(z) \leftarrow u \qquad h(\bar{z}) \leftarrow t[s[\bar{u}]].$$

Folding the first definition of f with the second definition of h yields the new definition

$$h(z) \leftarrow t[f(u)].$$

Unfolding the second definition of f now yields a fourth definition for h,

$$h(z) \leftarrow t[u[u]].$$

We have

$$h(z) \leftarrow r$$
$$h(z) \leftarrow t[s[u]]$$
$$h(z) \leftarrow t[f(\bar{u})]$$
$$h(z) \leftarrow t[u[u]].$$

Two of these definitions of h can be dropped, giving the final program,

$$f(\bar{z}) \Leftarrow s \qquad h(\bar{z}) \leftarrow r$$
$$f(z) \leftarrow u \qquad h(\bar{z}) \leftarrow t[u[\bar{u}]],$$

which represents the desired result.

In a similar fashion, the abstraction and composition transformations for expression procedures can be simulated. (Unfortunately, there seems to be no straightforward way to simulate our transformations on expression procedures so that Kott's restrictions are satisfied.)

8. The Deductive Approach

The systems of Wegbreit [Wegbreit76] and Manna and Waldinger [Manna79] are both based on a deductive, subgoaling approach. (Note that this discussion does not address the system of [Manna80], in which program derivation is viewed as a theorem-proving problem.)

In the deductive approach, the goal of developing a program for a particular computational task is split into various subgoals, which are analyzed

individually. Subgoals that are primitive can be computed directly. Subgoals that are derivative, on the other hand, must be analysed further, possibly by expanding a definition. When a derivative subgoal is found that matches a prior subgoal, or a generalization of a prior subgoal, then a recursive procedure can be formed.

We illustrate this using the vector product example. The style of this derivation is that of Manna and Waldinger, Wegbreit's derivations are in a similar form.

Vector product. The definition of the vector product p is specified as an *auxiliary transformation rule*,

$$p(a,b,n) \quad \Rightarrow \quad \text{if } n = 0 \text{ then } 0 \text{ else } a_n b_n + p(a,b,n-1),$$

and the Compute specification is specified as the top-level goal,

Goal 1 Compute $p(a,b,n) \Leftarrow p(c,d,n)$

We seek to develop a method for computing this goal. Since we may be developing a recursive program, we will write the goal as

Goal 1 Compute $f(a,b,c,d,n)$

with the auxiliary definition,

$$f(a,b,c,d,n) \Leftarrow \text{Compute } p(a,b,n) + p(c,d,n)$$

Now, if we ever encounter an instance of the Compute specification in the process of finding a program to satisfy the goal, we can generate a recursive call to the function f.

The first derivation step is to apply the transformation rule for p twice, to obtain the new goal,

Goal 2 Compute (if $n = 0$ then 0 else $a_n b_n + p(a,b,n-1)$)
(if $n = 0$ then 0 else $c_n d_n + p(c,d,n-1)$)

This is analogous to the unfolding step in the UF derivation. After simplification, the goal becomes

Goal 3: Compute: if $n = 0$ then 0
else $a_n b_n + c_n d_n + p(a, b, n-1) + p(c, d, n-1)$.

We now focus attention on the nonprimitive part of this goal,

Goal 4: Compute: $p(a, b, n-1) + p(c, d, n-1)$,

since the rest of the goal consists of primitive functions that we know how to compute.

This new goal is an instance of Goal 1, and so it is natural to form a recursion. We use the *recursion formation* rule to satisfy the goal by replacing it by a recursive call,

Goal 5: Compute: $f(a, b, c, d, n-1)$.

Since we already know how to compute goals of this form, no further subgoals need be generated. (In the implementation of this system, an automatic termination proof is done at this point, to ensure that the new recursive program always terminates.)

The tree of subgoals and their solutions is now reconstructed into a program for computing the initial goal.

$f(a, b, c, d, n) \leftarrow$ if $n = 0$ then 0
else $a_n b_n + c_n d_n + f(a, b, c, d, n-1)$

Discussion. In many examples, a later goal appears that is not a direct instance of any earlier goal, but is instead an instance of a generalization of an earlier goal. In this case, we attempt to develop an auxiliary procedure for the generalized version of the earlier goal by modifying the program for the initial goal into a more general form and repeating the derivation. If this is successful, then we can use the more general program to satisfy the initial goal. (See [Manna79] for a detailed description of the process. Generalization in a theorem-proving context is discussed in [Boyer75] and [Aubin75].)

For example, in a derivation of a *rev* program in [Manna79] generalization must be used to introduce a procedure *reversegen*(u, v), which has the effect of computing $rev(u) \circ v$ in the case that $u \neq \Lambda$. This procedure corresponds to the *rev*2 procedure that appears in our derivation.

In the context of the expression procedure system, the technique of generalization can be interpreted simply as a specialization heuristic, used to find new locations for boundaries in recursive functions. (See Chapter 3 for a description of boundary shifting.)

It is easy to see that the UF system underlies the deductive method. Observe that the recursion-formation rule corresponds directly to the fold rule of the UF system, and the application of auxiliary definitions corresponds to unfolding or folding the other definitions in a UF program. This correspondence can be shown in a more rigorous fashion by constructing a mapping from deductive derivations to UF derivations, such that every step of the deductive derivation maps to an application of one of the transformation rules or of a simplification rule.

This mapping indicates that it may be possible to apply Kott's restrictions directly to the deductive style derivations, since they have a similar form. At present Manna and Waldinger avoid the necessity for such restrictions by including a facility for automatic termination proofs in their implemented system. Similarly, Wegbreit's system guarantees termination by virtue of its use of automatic program analysis techniques.

9. Summary

Our system has concrete advantages over both of the UF based systems. The deductive systems (including [Manna80]) can correctly generate only programs which always terminate. Thus, these systems could not be used to derive parsing algorithms, as we do in Chapter 5, since some of these algorithms (e.g., top-down) do not terminate in all cases.

The system of Burstall and Darlington does not preserve strong equivalence in general, but can be restricted to do so. These restrictions, unfortunately, are on the *structure* of the derivations, not on the individual rules. In our system, the rules themselves preserve strong equivalence, and thus correctness considerations do not influence the structure of our derivations.

Chapter 5

Deriving Parsing Algorithms

Program derivation techniques are now being increasingly used as a way of explaining complicated algorithms. This approach is based on the thesis that a *derivation* of an algorithm reveals more to the intuition than does a conventional proof of correctness. Conventional proofs may succeed in convincing the reader of the correctness of an algorithm without giving him any hint of *why* the algorithm works or how it came about. On the other hand, a derivation may be thought of as an especially well-structured "constructive" proof of correctness of the algorithm, taking the reader step by step from an initial algorithm he accepts as a specification of the problem to a highly connected and efficient implementation of it.

In addition to the theoretical benefits, this approach has significant potential application to practical program maintainance, in particular to the processes of documentation and modification. Of course additional effort may be required to develop a complete initial derivation of a program; the potential savings in maintainance should more than offset this, however.

This approach to the description of algorithms has been explored previously by Dijkstra, Clark and Darlington, Green and Barstow, Correll, and others. (We have discussed these approaches in greater detail in other chapters.)

In this chapter, we show how the program derivation techniques developed in the previous chapters can be applied to the development of context-free parsing algorithms. All of these derivations start from the same initial specification of parsing, which is essentially a recursive definition of the *derives* relation of context-free languages.

The derivations will generally consist of a series of ***specialization*** steps. (Specialization is described in detail in Chapter 3.) While most of these steps specialize the algorithm in fairly routine ways, others require insight beyond that which might be expected of an automatic system. These "eureka" steps, however, tend to reveal more about the conceptual basis of the algorithm being developed than the other routine steps. The eureka step in the derivation of Earley's algorithm, for example, reveals how a top-down component can be added to a bottom-up algorithm.

The programs we derive in this chapter are more complicated than the examples given earlier in this work. For this reason, we need to use a slightly extended version of the language defined in Chapter 2. The extensions we introduce will enable us to manipulate the parsing programs with greater ease.

1. Relational Programs

All the programs derived in this chapter describe relations. In this section, we informally describe several extensions to the language defined in Chapters 2 and 3, in order to facilitate the development of relational programs.

Since a complete investigation of relational languages would be beyond the scope of this work, and since most of the concepts presented are well known, we keep the style in this section informal.

The use of relational languages in program development has been investigated in some depth in the context of PROLOG. (See e.g., [Kowalski79] and [Clark77].) Our approach to relations is somewhat different than this one, in that we are not bound to implement the full relational language described in this section, which includes parallel operations and bounded quantification. Such features, like expression procedures, are added to the language so that we can take advantage of their expressive power in the course of our derivations. Our intent is ultimately to transform the relational programs to use a simple subset of the language that can be easily implemented, such as the language of basic programs described in Chapter 2 (or the language of forward programs outlined in the Appendix.)

The Propositional Connectives. In relational programs, control structuring is provided by the propositional connectives. In this section, we discuss several different meanings that can be given to the propositional connectives in the context of programming, and describe the particular meanings we use in our examples. In general, when using the propositional operations in a program, it is necessary to specify the results of these operations for the special case in which evaluation of one or both of the arguments loops.

In LISP, for example, the propositional connectives have the property that they are *sequential*. In an evaluation of $s \wedge t$, if s evaluates to **false**, then t will not be evaluated, since the result of the conjunction has already been determined to be **false**. Let *loop* stand for a term whose evaluation never terminates. Then in LISP we have

$$\textbf{false} \wedge \mathit{loop} = \textbf{false} \qquad \text{and} \qquad \textbf{true} \wedge \mathit{loop} = \mathit{loop},$$

On the other hand, $\mathit{loop} \wedge t$ for any term t results in a loop. We call this LISP method of evaluation the *ordered sequential* interpretation.

A more conservative approach would be to initially evaluate *both* arguments of a propositional operator, and then calculate the result. (This is called a *strict* interpretation.) In this case, $s \wedge t$ would result in a loop whenever either s or t loops, regardless of the value of the the other term. That is,

$$s \wedge \mathit{loop} \;=\; \mathit{loop} \wedge s \;= \mathit{loop},$$

and similarly for disjunction. We call this the *unordered sequential* interpretation of the operators. Under this interpretation, the propositional operators cannot be used to regulate the flow of control, unlike the ordered sequential LISP operators. (See [McCarthy78].)

There is a third approach to interpreting the propositional connectives, which is called the *parallel* interpretation. This interpretation is best described operationally as follows: To evaluate $s \vee t$, start evaluating the terms s and t together in *parallel*. As soon as evaluation of one of the terms terminates, examine its value. If the value is **true**, then halt evaluation of the other argument and return **true** as the result; otherwise, wait until evaluation of the other argument terminates and return its result as the value of the expression. Thus, we have

$$s \vee \textbf{true} \;=\; \textbf{true} \vee s \;= \textbf{true},$$

even if evaluation of s loops. Because parallel evaluation is required to implement such operations, the evaluator models presented in Chapter 2, which are all sequential, are inadequate.

Note that the usual **if then else** conditional is equivalent to a combination of propositional connectives:

$$\textbf{if } P \textbf{ then } Q \textbf{ else } R \;=\; (P \wedge Q) \;\vee\; (\neg P \wedge R),$$

where $\wedge$ is sequential. Conditionals do not require parallelism to implement because of the special role of the term P: After P is evaluated, only one of the two disjuncts can possibly be true, so only that disjunct need be evaluated.

In our programs we will use two of these propositional connectives: sequential conjunction, written $\wedge$, and parallel disjunction, written $\vee$. (We use a smaller symbol for conjunction to indicate that it has a higher syntactic precedence than disjunction.) Ultimately, we must transform the programs involving these operations into programs using only ordinary (sequential) conditionals, since unbounded parallelism is not practical to implement. Our use of parallelism, however, frees us from having to deal explicitly with nondeterminism in the initial part of the development process.

Bounded Existential Quantification. In addition to the propositional connectives, we will make use of bounded existential quantification. This gives us a means of introducing new auxiliary variables, but does not increase the power of the language, since a bounded existential quantification can always be expanded into a finite disjunction. (The notation mechanism could be used to effect this.)

We use the notation $[\exists x \mid P(x)]$ to indicate that x is an existential variable whose domain is bounded by the predicate P.

An example of a relation with existentially quantified variables is the *Fibonacci relation* F defined over the nonnegative integers, where $F(i, n)$ holds if n is the ith Fibonacci number.

$$\begin{aligned} F(i,n) \leftarrow \quad & i \leq 1 \;\wedge\; i = n \quad \vee \\ & i > 1 \;\wedge\; [\exists u, v \mid 0 \leq u, v \leq n] \\ & \qquad F(i-1,u) \;\wedge\; F(i-2,v) \;\wedge\; u + v = n \end{aligned}$$

Thus, $F(0,0)$, $F(1,1)$, $F(2,1)$, $F(3,2)$, $F(4,3)$, $F(5,5)$, $F(6,8)$, etc.

A bound on an existential variable is a *conservative bound* if the existentially quantified term is false whenever the value of the existential variable falls outside the bound. For example, the bounds on the existential variables u and v in the Fibonacci program above are conservative.

In fact, the bounds in our examples will always be conservative bounds, and so we will usually omit bounds specifications entirely since they are generally obvious from context. Furthermore, if we assume that all variables that occur in the body part of a definition but not in the name part are existentially quantified, then we will be able to omit most of the existential quantifiers from our programs. The bounds in the Fibonacci definition above, for example, are conservative. Thus, we can write the definition in the abbreviated form:

$$\begin{aligned} F(i,n) \leftarrow \quad & i \leq 1 \ \wedge \ i = n \quad \vee \\ & i > 1 \ \wedge \ F(i-1,u) \ \wedge \ F(i-2,v) \ \wedge \ u+v = n. \end{aligned}$$

In the course of simplification, we naturally try to eliminate existential variables wherever possible. For example, the expression

$$[\exists z]\,(z = s \ \wedge \ P(z)),$$

where the term s contains no instances of z, can be replaced by the simpler expression $P'(s)$. Similarly, $[\exists z]\,(z = s)$ can be trivially simplified to **true**. We can also uniformly rename existential variables in a term; in some instances (particularly when using the application rule), such renaming may be necessary to prevent name conflicts. In addition, we will make use of other standard rules for manipulating existential quantifiers, such as distribution over disjunction,

$$[\exists z]\,(P(z) \vee Q(z)) \quad = \quad [\exists z]\,P(z) \ \vee \ [\exists z]\,Q(z).$$

The application rule can be used on an existentially quantified subterm in two ways. Suppose we are simplifying a definition whose body contains the subterm $[\exists z]\,f(z,t)$. Then the definition $f(x,y) \leftarrow E[x,y]$ can be unfolded in the usual way, yielding $[\exists z]\,E[z,t]$. On the other hand, we could unfold an

expression procedure defining the entire existentially quantified expression. For example, we could unfold the expression procedure $[\exists x]\,f(x,y) \leftarrow E'[y]$ to obtain the term $E'[t]$.

It is often advantageous to form expression procedures for existentially quantified terms. We use a special composition rule for doing this:

$$\text{given} \quad f(x,\bar{z}) \leftarrow E, \qquad \text{add} \quad [\exists x]f(x,\bar{z}) \leftarrow [\exists x]E$$

(This rule can be applied only if there are conservative bounds for x; i.e., $f(x,\bar{z})$ is true for only a finite number of values of x and the set is effectively computable.) The use of this rule often permits significant simplification in the new definition body. For example, we could form an expression procedure for the term $[\exists i]F(i,n)$, where F is the Fibonacci function defined above. This term is equivalent to **true** if n is a Fibonacci number. Using the composition rule, we obtain

$$\begin{aligned} [\exists i]F(i,n) \leftarrow \quad & [\exists i]\, i \leq 1 \ \wedge \ i = n \quad \vee \\ & [\exists i]\, i > 1 \ \wedge \ F(i-1,u) \ \wedge \ F(i-2,v) \ \wedge \ u+v = n. \end{aligned}$$

(Note that there are conservative bounds for i; e.g., $0 \leq i \leq n$.) This definition simplifies to

$$\begin{aligned} [\exists i]F(i,n) \leftarrow \quad & n \leq 1 \quad \vee \\ & [\exists i]\, i > 1 \ \wedge \ F(i-1,u) \ \wedge \ F(i-2,v) \ \wedge \ u+v = n. \end{aligned}$$

We could obtain a more successful simplification if we were interested in testing $[\exists n]F(i,n)$, which is **true** if there is an ith Fibonacci number. Since the domain of i is the nonnegative integers, composition and induction eventually yield

$$[\exists n]F(i,n) \leftarrow \quad \textbf{true}.$$

Typed variables. In the parser examples there will be variables ranging over several different domains, such as symbols, strings, and the nonnegative integers. In a more rigorous development we would need to introduce qualifiers for typed variables to ensure that they range only over the appropriate domains. In this presentation, however, we consider such qualifiers to be implicit, and instead make use of fonts to remind us of the types of variables. (An alternative is to extend the language to include explicit typing of variables, but this seems more cumbersome than our use of implicit qualifiers. Note that this mechanism of type qualifiers provides a useful method for developing specialized versions of "generic" procedures.)

Errors Certain common primitive functions are only partially defined over the standard domains such as strings or nonnegative integers. The list selection functions *hd* and *tl*, for example, are defined over all strings ***except*** the null string Λ. Rather than extending these functions to total functions in an arbitrary way, or requiring (through the use of qualifiers) that appropriate preconditions are met, we instead consider "illegal" terms to have a special value, **error**, and we introduce conventions for handling this special value. Thus, the value of terms such as $hd(\Lambda)$, $tl(3)$, and $2 - 5$ is the special value **error**.

The conventions for handling this value are very simple: If the value of any strict argument to a function is **error**, then the result will be **error** (so derivative functions are **error**-strict). Further, if the value of any argument to a relation is **error**, then the evaluator will return **false** as the result of that relation. (This unusual convention regarding relations is justified in the Appendix.)

Thus according to this convention, we could omit the test $i > 1$ in the body of our Fibonacci relation and write only

$$F(i,n) \leftarrow \quad i \leq 1 \wedge i = n \quad \vee$$
$$F(i-1,u) \wedge F(i-2,v) \wedge u + v = n.$$

When $i \leq 1$ both of the recursive calls will evaluate to **false**, so this definition is equivalent to the previous one.

Imperative programs. In all of the examples in this chapter (and in the previous chapters) we manipulate recursive functions defined in an applicative language. This approach allows for powerful and elegant transformation rules (as has been shown in e.g., [Burstall77] and [Manna79]) but it does not permit us to reason easily about critical implementation considerations such as reuse of storage.

For this reason, it is implicit in our approach that the recursive programs we derive be further developed into programs using imperative constructs. We discuss several methods for doing this in the Appendix.

2. Parsing

The initial specification we take for the parsing process is a recursive definition of the *derives* relation $\alpha \stackrel{*}{\Rightarrow} \beta$, where α and β are strings. The parsing problem can then be stated:

$$\underline{\text{Compute}}: \quad [S] \stackrel{*}{\Rightarrow} [a_1, \ldots, a_M],$$

where S is the "start" symbol, and $[a_1, \ldots, a_M]$ is a terminal ***final string***.

Grammars. More precisely, let $G = (V, \Sigma, P, S)$ be a context-free grammar. Here, the *vocabulary* of symbols V includes the set of ***terminal*** symbols Σ as a subset. The set P of ***productions*** is a finite subset of $N \times V^*$, and the symbol S is an element of the set of ***nonterminal*** symbols $N = V - \Sigma$. In our examples, the types of variables are generally indicated by their names:

$a, b, c, \ldots \in \Sigma$	(terminals)
$A, B, C, \ldots \in N$	(nonterminals)
$\rho, \sigma, \psi, \ldots \in V$	(symbols)
$\alpha, \beta, \gamma, \ldots \in V^*$	(strings)

The production relation $P(A, \beta)$ may be written as

$$A \rightarrow \beta.$$

(For additional background on context-free grammars and parsing algorithms see [Aho73] and [Hopcroft69].)

Strings. Several operations on strings are defined: If $\sigma_1, \sigma_2, \ldots, \sigma_n$ are elements of V, then an explicit string consisting of these elements is written $[\sigma_1, \sigma_2, \ldots, \sigma_n]$. This describes a string whose length is n. The explicit strings $[]$ and $[\sigma_i, \ldots, \sigma_{i-1}]$ both denote the empty string Λ. Strings can be constructed from other strings using the ***append*** operation, which concatenates two strings: Thus,

$$[\sigma_1, \ldots, \sigma_n] \circ [\rho_1, \ldots, \rho_m] = [\sigma_1, \ldots, \sigma_n, \rho_1, \ldots, \rho_m].$$

We define four operations for selecting elements or parts of strings. These operations are all undefined on the empty string (but see note in the previous

section).

$$\begin{aligned} hd([\sigma_1, \sigma_2, \ldots, \sigma_{n-1}, \sigma_n]) &= \sigma_1 \\ tl([\sigma_1, \sigma_2, \ldots, \sigma_{n-1}, \sigma_n]) &= [\sigma_2, \ldots, \sigma_{n-1}, \sigma_n] \\ fr([\sigma_1, \sigma_2, \ldots, \sigma_{n-1}, \sigma_n]) &= [\sigma_1, \sigma_2, \ldots, \sigma_{n-1}] \\ lt([\sigma_1, \sigma_2, \ldots, \sigma_{n-1}, \sigma_n]) &= \sigma_n \end{aligned}$$

Thus, if α is a nonempty string,

$$[hd(\alpha)] \circ tl(\alpha) = \alpha, \quad \text{and} \quad fr(\alpha) \circ [lt(\alpha)] = \alpha.$$

The parsing problem. The initial specification we use for the parsing problem is phrased in terms of the relation D defined as follows.

$$\begin{aligned} D(\psi, \pi) \leftarrow \quad & [\psi] = \pi \quad \vee \\ & \psi \rightarrow \beta \ \wedge \ \beta = [\sigma_1, \ldots, \sigma_n] \ \wedge \\ & \quad D(\sigma_1, \delta_1) \ \wedge \ \cdots \ \wedge \ D(\sigma_n, \delta_n) \ \wedge \\ & \quad \delta_1 \circ \cdots \circ \delta_n = \pi \end{aligned}$$

The relation $D(\psi, \pi)$, where ψ is a vocabulary symbol and π is a string, defines a restriction of the usual *derives* relation. According to the definition, the symbol ψ derives π if they are equivalent, or if there is a production from ψ to a string β each of whose elements derives some portion of the string π. Thus $D(\psi, \pi)$ is equivalent to $[\psi] \stackrel{+}{\Rightarrow} \pi$.

Thus, our goal is to test

$$D(S, [a_1, \ldots, a_M]).$$

We will accept this simple definition of a restriction of the "$\stackrel{+}{\Rightarrow}$" relation as our starting point for the parsing algorithm derivations.

We make several notes about this definition: First, we will assume implicit type qualifiers restricting the variables to the types indicated by their fonts. Thus, ψ must be a *single* vocabulary *symbol*, and π must be a *string* of vocabulary symbols.

Second, observe that we are making use of the ellipsis notation defined in Chapter 3.

Third, we note that the variables n, $\sigma_1, \ldots, \sigma_n$, and $\delta_1, \ldots, \delta_n$ are all implicitly existentially quantified over the definition body. The variable n can be bounded by the maximum length of the right-hand side in the production relation P. Since the vocabulary is finite, the range of the variables $\sigma_1, \ldots, \sigma_n$ is clearly bounded. Finally, the range of the variables $\delta_1, \ldots, \delta_n$ can be bounded to the set of strings of length less than or equal to than of the string π.

The final note we make is that not all computation paths of D terminate. (Consider the case in which there is a production of the form $S \rightarrow S$.) There may be more than one computation path for a given grammar and final string, however, and we note that if there is a parse, then there is a successful terminating computation path for D. While this existential termination property is characteristic of many parsing algorithms (such as the top-down algorithm), other algorithms (such as the Cocke-Younger-Kasami algorithm and Earley's algorithm) are universally terminating. Because of these differences, we postpone discussion of termination until the developments of the individual algorithms. This issue is also discussed in the Appendix.

From this definition we will derive several parsing algorithms. We give a simple example derivation before proceeding to the main derivations.

3. A Highly Specialized Parser

It will be of value to work through a simple specialization example in some detail before tackling the more complicated parsing algorithms. We start in this example with the *derives* definition just presented.

$$\begin{aligned} D(\psi, \pi) \leftarrow \quad & [\psi] = \pi \quad \vee \\ & \psi \rightarrow \beta \ \wedge \ \beta = [\sigma_1, \ldots, \sigma_n] \ \wedge \\ & \quad D(\sigma_1, \delta_1) \ \wedge \ \cdots \ \wedge \ D(\sigma_n, \delta_n) \ \wedge \\ & \quad \delta_1 \circ \cdots \circ \delta_n = \pi \end{aligned}$$

Compute: $D(S, [a_1, \ldots, a_M])$

We will specialize this general parsing program to the grammar:

$$S \rightarrow aS \quad \text{or} \quad S \rightarrow \Lambda,$$

where S is the start symbol, a is the only terminal symbol, and Λ is the empty string. The language described by this grammar is the set of finite length strings of a's $\{a^i \mid i \geq 0\}$

Representing the production relation. Since the grammar has been completely specified, we can give an explicit definition for the production relation,

$$(\psi \rightarrow \beta) \quad \leftarrow \quad \psi = S \ \wedge \ (\beta = [a, S] \vee \beta = [])\,,$$

thus promoting it from a primitive function to a derivative function. We unfold this definition in the body of D,

$$\begin{aligned} D(\psi, \pi) \leftarrow \quad & [\psi] = \pi \quad \vee \\ & \psi = S \ \wedge \ (\beta = [a, S] \quad \vee \quad \beta = []) \ \wedge \\ & \quad \beta = [\sigma_1, \ldots, \sigma_n] \ \wedge \\ & \quad D(\sigma_1, \delta_1) \ \wedge \ \cdots \ \wedge \ D(\sigma_n, \delta_n) \ \wedge \\ & \quad \delta_1 \circ \cdots \circ \delta_n = \pi, \end{aligned}$$

and distribute the disjunction through,

$$\begin{aligned} D(\psi, \pi) \leftarrow \quad & [\psi] = \pi \quad \vee \\ & \psi = S \ \wedge \ \beta = [a, S] \ \wedge \ \beta = [\sigma_1, \ldots, \sigma_n] \ \wedge \\ & \quad D(\sigma_1, \delta_1) \ \wedge \ \cdots \ \wedge \ D(\sigma_n, \delta_n) \ \wedge \\ & \quad \delta_1 \circ \cdots \circ \delta_n = \pi \quad \vee \\ & \psi = S \ \wedge \ \beta = [] \ \wedge \ \beta = [\sigma_1, \ldots, \sigma_n] \ \wedge \\ & \quad D(\sigma_1, \delta_1) \ \wedge \ \cdots \ \wedge \ D(\sigma_n, \delta_n) \ \wedge \\ & \quad \delta_1 \circ \cdots \circ \delta_n = \pi. \end{aligned}$$

By application of the appropriate ellipsis definitions and simplifying, we obtain the more concise definition

$$\begin{aligned} D(\psi, \pi) \leftarrow \quad & [\psi] = \pi \quad \vee \\ & \psi = S \ \wedge \ D(a, \delta_1) \ \wedge \ D(S, \delta_2) \ \wedge \\ & \quad \delta_1 \circ \delta_2 = \pi \quad \vee \\ & \psi = S \ \wedge \ \Lambda = \pi. \end{aligned}$$

(We present a more detailed example of manipulation involving ellipses in the section on left-to-right parsers.)

Specialization: ψ is the terminal symbol a. The result of this last derivation step is that the two recursive calls to D are in specialized forms. We can develop specialized versions of D for these two cases. In the first case $\psi = a$; we therefore compose the definition of D (substituting a for ψ) to get

$$\begin{aligned} D(a, \pi) \leftarrow \quad & [a] = \pi \quad \vee \\ & a = S \ \wedge \ D(a, \delta_1) \ \wedge \ D(S, \delta_2) \ \wedge \\ & \quad \delta_1 \circ \delta_2 = \pi \quad \vee \\ & a = S \ \wedge \ \Lambda = \pi, \end{aligned}$$

which (since $a = S$ is false) immediately simplifies to

$$D(a, \pi) \leftarrow \quad [a] = \pi.$$

Since this definition is not recursive, we unfold it in the body of the definition of D and eliminate it. Our main program thus becomes

$$\begin{aligned} D(\psi, \pi) \leftarrow \quad & [\psi] = \pi \quad \vee \\ & \psi = S \ \wedge \ [a] = \delta_1 \ \wedge \ D(S, \delta_2) \ \wedge \\ & \quad \delta_1 \circ \delta_2 = \pi \quad \vee \\ & \psi = S \cdot \wedge \ \Lambda = \pi, \end{aligned}$$

which simplifies to

$$\begin{aligned} D(\psi, \pi) \leftarrow \quad & [\psi] = \pi \quad \vee \\ & \psi = S \ \wedge \ D(S, \delta) \ \wedge \ [a] \circ \delta = \pi \quad \vee \\ & \psi = S \ \wedge \ \Lambda = \pi \end{aligned}$$

Compute: $D(S, [a_1, \ldots, a_M])$.

(We eliminated the existential variable δ_1 and renamed the existential variable δ_2 to δ.)

Specialization: ψ is the start symbol. We observe now that all calls to D have the start symbol S as their first argument. We naturally specialize to take advantage of this. Composition (substituting S for ψ) yields

$$\begin{aligned} D(S, \pi) \leftarrow \quad & [S] = \pi \quad \vee \\ & S = S \ \wedge \ D(S, \delta) \ \wedge \ [a] \circ \delta = \pi \quad \vee \\ & S = S \ \wedge \ \Lambda = \pi, \end{aligned}$$

which simplifies to

$$\begin{aligned} D(S,\pi) \leftarrow \quad & [S]=\pi \quad \vee \\ & D(S,\delta) \wedge [a]\circ\delta=\pi \quad \vee \\ & \Lambda=\pi \\ \underline{\text{Compute}}: \quad & D(S,[a_1,\ldots,a_M]). \end{aligned}$$

Since S is constant over an execution of the program, we can leave it as an implicit global parameter. Abstraction yields

$$D(S,\pi) \leftarrow \quad D'(\pi)$$

with the new definition

$$\begin{aligned} D'(\pi) \leftarrow \quad & [S]=\pi \quad \vee \\ & D(S,\delta) \wedge [a]\circ\delta=\pi \quad \vee \\ & \Lambda=\pi. \end{aligned}$$

Unfolding the calls to D in this definition and in the Compute specification lets us eliminate the definition of D, leaving only

$$\begin{aligned} D'(\pi) \leftarrow \quad & [S]=\pi \quad \vee \\ & D'(\delta) \wedge [a]\circ\delta=\pi \quad \vee \\ & \Lambda=\pi. \\ \underline{\text{Compute}}: \quad & D'([a_1,\ldots,a_M]). \end{aligned}$$

Specialization: π is a final string suffix. We can specialize this definition even further, based on the hypothesis that π is always a suffix of the final string $[a_1,\ldots,a_M]$. We can represent a suffix of the final string using the string specification $[a_i,\ldots,a_M]$, where $i \le M+1$. Using composition to instantiate the definition of D' above, we obtain

$$\begin{aligned} D'([a_i,\ldots,a_M]) \leftarrow \quad & [S]=[a_i,\ldots,a_M] \quad \vee \\ & D'(\delta) \wedge [a]\circ\delta=[a_i,\ldots,a_M] \quad \vee \\ & \Lambda=[a_i,\ldots,a_M]. \\ \underline{\text{Compute}}: \quad & D'([a_1,\ldots,a_M]) \end{aligned}$$

(We leave the qualifier $(i \le M+1)$ on i implicit.)

Using the properties of strings, this simplifies to

$$\begin{aligned} D'([a_i,\ldots,a_M]) \leftarrow \quad & S=a_i \wedge i=M \quad \vee \\ & D'([a_{i+1},\ldots,a_M]) \wedge a=a_i \quad \vee \\ & i=M+1. \end{aligned}$$

The first disjunct is always false, so we have only

$$\begin{aligned} D'([a_i,\ldots,a_M]) \leftarrow \quad & a_i=a \wedge D'([a_{i+1},\ldots,a_M]) \quad \vee \\ & i=M+1, \end{aligned}$$

which is recursive, so the specialization is successful.

We therefore abstract and unfold to replace this definition by a basic one. The final string suffix specification $[a_i,\ldots,a_M]$ can be described in terms of the two parameters i and $[a_1,\ldots,a_M]$. While the value of i changes on every iteration of D', the string $[a_1,\ldots,a_M]$ remains constant throughout execution of the program, so we will leave it as an implicit second parameter.

We will call the new function formed by abstraction R. Abstraction yields

$$\begin{aligned} & D'([a_i,\ldots,a_M]) \leftarrow \quad R(i) \\ & R(i) \leftarrow \quad a_i=a \wedge D'([a_{i+1},\ldots,a_M]) \quad \vee \quad i=M+1 \\ & \underline{\text{Compute}}: \quad D'([a_1,\ldots,a_M]). \end{aligned}$$

Unfolding the two calls to D' and eliminating that definition results in the very simple linear time program,

$$\begin{aligned} & R(i) \leftarrow \quad a_i=a \wedge R(i+1) \quad \vee \quad i=M+1. \\ & \underline{\text{Compute}}: \quad R(1). \end{aligned}$$

(The techniques used in this derivation could be used to specialize D to the more general case in which P is a regular grammar; that is in which all productions in P are of the form

$$A \rightarrow aB, \qquad \text{or} \qquad A \rightarrow a,$$

where a is terminal and A and B are both nonterminal.)

4. The Cocke-Younger-Kasami Algorithm

The recursive version of the Cocke-Younger-Kasami algorithm arises very naturally from the initial definition for D simply by restricting the production relation P such that all productions are in Chomsky Normal Form.

Recall that a grammar is in *Chomsky Normal Form (CNF)* if all of its productions have the form

$$A \to a \qquad \text{or} \qquad A \to BC,$$

where A, B, and C are nonterminal symbols and a is terminal. Thus, for the purpose of simplification, we can use the fact that

$$\psi \in N \;\wedge\; \big(\exists a(\beta = \lfloor a \rfloor) \;\vee\; \exists B, C(\beta = \lfloor B, C \rfloor)\big).$$

holds whenever $\psi \to \beta$ holds.

Observe that evaluations of D always terminate when the grammar is in Chomsky Normal Form.

CNF specialization In the first step of the derivation we simplify the body of the definition of D using the CNF property of the production relation. We first introduce additional (primitive) conjuncts to reflect the Chomsky Normal Form assumption:

$$\begin{aligned} D(\psi,\pi) \leftarrow\ & \lfloor\psi\rfloor = \pi \;\vee \\ & \psi \to \beta \;\wedge\; \psi \in N \;\wedge\; (\beta = \lfloor a \rfloor \;\vee\; \beta = \lfloor B, C \rfloor) \;\wedge \\ & \quad \beta = \lfloor \sigma_1, \ldots, \sigma_n \rfloor \;\wedge \\ & \quad D(\sigma_1, \delta_1) \;\wedge\; \ldots \;\wedge\; D(\sigma_n, \delta_n) \;\wedge \\ & \quad \delta_1 \circ \cdots \circ \delta_n = \pi \end{aligned}$$

To simplify this, we distribute the new disjunction through, introducing a case split,

$$\begin{aligned} D(\psi,\pi) \leftarrow\ & \lfloor\psi\rfloor = \pi \;\vee \\ & \psi \to \beta \;\wedge\; \psi \in N \;\wedge\; \beta = \lfloor a \rfloor \;\wedge\; \beta = \lfloor \sigma_1, \ldots, \sigma_n \rfloor \;\wedge \\ & \quad D(\sigma_1, \delta_1) \;\wedge\; \ldots \;\wedge\; D(\sigma_n, \delta_n) \;\wedge \\ & \quad \delta_1 \circ \cdots \circ \delta_n = \pi \;\vee \\ & \psi \to \beta \;\wedge\; \psi \in N \;\wedge\; \beta = \lfloor B, C \rfloor \;\wedge\; \beta = \lfloor \sigma_1, \ldots, \sigma_n \rfloor \;\wedge \\ & \quad D(\sigma_1, \delta_1) \;\wedge\; \ldots \;\wedge\; D(\sigma_n, \delta_n) \;\wedge \\ & \quad \delta_1 \circ \cdots \circ \delta_n = \pi, \end{aligned}$$

and simplify using the properties of strings,

$$\begin{aligned} D(\psi,\pi) \leftarrow\ & \lfloor\psi\rfloor = \pi \;\vee \\ & \psi \to a \;\wedge\; \psi \in N \;\wedge\; D(a,\pi) \;\vee \\ & \psi \to \lfloor B, C \rfloor \;\wedge\; \psi \in N \;\wedge \\ & \quad D(B, \delta_1) \;\wedge\; D(C, \delta_2) \;\wedge\; \delta_1 \circ \delta_2 = \pi. \end{aligned}$$

(In this step, we eliminated the ellipsis notation by unfolding appropriate ellipsis expression procedure schemas.)

All three of the recursive calls to D now have the property that the type of the first argument (terminal or nonterminal symbol) is known in advance. It is therefore natural to specialize D to these two cases.

Specialization: ψ is a terminal symbol. In the first case, we substitute a terminal symbol variable c for the general variable ψ:

$$\begin{aligned} D(c,\pi) \leftarrow\ & \lfloor c \rfloor = \pi \;\vee \\ & c \to a \;\wedge\; c \in N \;\wedge\; D(a,\pi) \;\vee \\ & c \to \lfloor B, C \rfloor \;\wedge\; c \in N \;\wedge \\ & \quad D(B, \delta_1) \;\wedge\; D(C, \delta_2) \;\wedge\; \delta_1 \circ \delta_2 = \pi. \end{aligned}$$

The term $c \in N$ simplifies to false, so the second and third disjuncts drop out, leaving the very simple nonrecursive definition,

$$D(c,\pi) \leftarrow \lfloor c \rfloor = \pi.$$

Specialization: ψ is a nonterminal symbol. In this second case, we form an expression procedure for the case in which ψ is a nonterminal symbol A. Since the Compute specification has this form, a successful specialization will let us eliminate the more general version of D. The result of composition is

$$\begin{aligned} D(A,\pi) \leftarrow\ & \lfloor A \rfloor = \pi \;\vee \\ & A \to a \;\wedge\; A \in N \;\wedge\; D(a,\pi) \;\vee \\ & A \to \lfloor B, C \rfloor \;\wedge\; A \in N \;\wedge \\ & \quad D(B, \delta_1) \;\wedge\; D(C, \delta_2) \;\wedge\; \delta_1 \circ \delta_2 = \pi. \end{aligned}$$

The only recursive call to D that is not in the specialized form is the one in the second disjunct. To eliminate it, we simply apply the nonrecursive expression procedure derived in the last step.

$$\begin{aligned} D(A,\pi) \leftarrow \quad & [A]{=}\pi \quad \vee \\ & A \rightarrow a \wedge [a]{=}\pi \quad \vee \\ & A \rightarrow [B,C] \wedge D(B,\delta_1) \wedge D(C,\delta_2) \wedge \delta_1 \circ \delta_2 {=} \pi \end{aligned}$$

Compute: $D(S,[a_1,\ldots,a_m])$

(We dropped the test $A \in N$ since it is always **true**.) All calls to D are now in the specialized form, so the specialization is successful. Renaming is unnecessary because of our use of typed variables.

This last step of specialization came about because of observations we made about the form of the first argument to D. Further specializations can be made on the basis of observations about the form of the second argument, π.

Specialization: π is a substring of the final string. The next step of specialization is based on the hypothesis that the second argument π of D is always a substring of the final string.

We compose D by substituting for π a specification of an arbitrary substring of the final string. There are several natural ways to specify a substring of a given string $[\rho_1,\ldots,\rho_n]$. One is to distinguish the starting and ending characters in the string, writing $[\rho_i,\ldots,\rho_j]$. Another is to distinguish only the starting point and length of the string as in $[\rho_i,\ldots,\rho_{i+\ell-1}]$. Here, i is the starting point and ℓ is the length of the substring. We choose the second form, and thus obtain

$$\begin{aligned} D(A,[a_i,\ldots,a_{i+\ell-1}]) \leftarrow \quad & [A]{=}[a_i,\ldots,a_{i+\ell-1}] \quad \vee \\ & A \rightarrow a \wedge [a]{=}[a_i,\ldots,a_{i+\ell-1}] \quad \vee \\ & A \rightarrow [B,C] \wedge D(B,\delta_1) \wedge D(C,\delta_2) \wedge \\ & \quad \delta_1 \circ \delta_2 {=} [a_i,\ldots,a_{i+\ell-1}]. \end{aligned}$$

Because of our error convention, we need not consider cases in which ℓ is negative or in which the substring falls outside the limits of the original final string.

We now simplify this new definition using the usual properties of strings and integers.

$$\begin{aligned} D(A,[a_i,\ldots,a_{i+\ell-1}]) \leftarrow \quad & [A]{=}[a_i] \wedge \ell{=}1 \quad \vee \\ & A \rightarrow a \wedge [a]{=}[a_i] \wedge \ell{=}1 \quad \vee \\ & A \rightarrow [B,C] \wedge D(B,[a_i,\ldots,a_{i+k-1}]) \wedge \\ & \quad D(C,[a_{i+k},\ldots,a_{i+\ell-1}]) \end{aligned}$$

The crucial step here was the elimination of the *append* operation in favor of a new existential variable k. Because of our error conventions, we can bound quantification of k to the interval $0 \le k \le \ell$.

The result of this simplification is that all calls to D are in the new specialized form. Before renaming, we make two more simplifications: First, type restrictions prevent $[A] = [a_i]$ from ever being true, so we can drop the first disjunct. Second, we eliminate the existential variable a from the second disjunct since it is always equal to a_i.

$$\begin{aligned} D(A,[a_i,\ldots,a_{i+\ell-1}]) \leftarrow \quad & A \rightarrow a_i \wedge \ell{=}1 \quad \vee \\ & A \rightarrow [B,C] \wedge D(B,[a_i,\ldots,a_{i+k-1}]) \wedge \\ & \quad D(C,[a_{i+k},\ldots,a_{i+\ell-1}]) \end{aligned}$$

Compute: $D([S],[a_1,\ldots,a_M])$

Since all calls to D are now in the specialized form, we can rename this procedure to avoid the explicit specification of the substrings. We abstract the parameter $[a_i,\ldots,a_{i+\ell-1}]$ into a set of three parameters: the integers i and ℓ, and the complete final string $[a_1,\ldots,a_M]$. In this example, however, we will take the final string as an *implicit* global variable, to avoid carrying a constant parameter through the entire computation. After abstraction and application (and renaming the new name back to D) our program becomes

$$\begin{aligned} D(A,i,\ell) \leftarrow \quad & A \rightarrow a_i \wedge \ell{=}1 \quad \vee \\ & A \rightarrow [B,C] \wedge D(B,i,k) \wedge D(C,i{+}k,\ell{-}k) \wedge 0{\le}k{\le}\ell \end{aligned}$$

Compute: $D(S,1,M)$

Induction: reducing the bounds on k We now make a rather subtle improvement to the program, using computation induction. Recall that *Chomsky Normal Form* grammars have the property that the language they describe contains no empty strings. This fact should enable us to specialize our program to the case where $\ell \neq 0$. We will need to use induction to achieve this simplification, since we need to draw information out from the inner recursive calls, rather than bringing it inward, as in our usual specializations. That is, we need to make use of the fact that $\ell \neq 0$ in the base case in order to achieve the simplification.

In this induction, the induction hypothesis is simply

$$D(A,i,\ell) \supset \ell \neq 0.$$

It is convenient in this presentation (and in most relational developments) to leave the implication implicit and write the induction hypothesis in a qualifier simply as $\ell \neq 0$. We start by assuming the induction hypothesis for the two recursive calls to D:

$$\begin{aligned} D(A,i,\ell) \leftarrow \quad & A \to a_i \wedge \ell = 1 \quad \vee \\ & A \to [B,C] \wedge \{k \neq 0\}\, D(B,i,k) \wedge \\ & \quad \{\ell - k \neq 0\}\, D(C,i+k,\ell-k) \wedge 0 \leq k \leq \ell. \end{aligned}$$

Properties of the integers yield the body qualifier $\{\ell > 1\}$ for the second disjunct, and $\{\ell = 1\}$ obviously holds in the first. Since

$$\ell > 1 \quad \vee \quad \ell = 1 \quad \supset \quad \ell \neq 0,$$

the induction is successful, and we can introduce $\{\ell \neq 0\}$ as an assertion which holds whenever $D(A,i,\ell)$ does.

$$\begin{aligned} D(A,i,\ell)\ \{\ell \neq 0\} \leftarrow \quad & A \to a_i \wedge \ell = 1 \quad \vee \\ & A \to [B,C] \wedge \{k \neq 0\}\, D(B,i,k) \wedge \\ & \quad \{\ell - k \neq 0\}\, D(C,i+k,\ell-k) \wedge 0 \leq k \leq \ell \end{aligned}$$

We can therefore use the qualifiers $\{k \neq 0\}$ and $\{\ell - k \neq 0\}$ in D to restrict the inequality $0 \leq k \leq \ell$ to $0 < k < \ell$. We obtain the program

$$\begin{aligned} D(A,i,\ell) \leftarrow \quad & A \to a_i \wedge \ell = 1 \quad \vee \\ & A \to [B,C] \wedge D(B,i,k) \wedge D(C,i+k,\ell-k) \wedge 0 < k < \ell \end{aligned}$$

Compute: $D(S,1,M)$.

We now have a recursive version of the Cocke-Younger-Kasami algorithm. We can describe the action of this algorithm as follows: $D(A,i,\ell)$ is true if there is a derivation from the nonterminal symbol A to the portion of the final string that starts at position i and has length ℓ. There is such a derivation if either of the following two conditions hold: The first condition, corresponding to the first disjunct, is that this portion of the final string consists of a single symbol, and there is a production from A to this symbol. The second condition is that there is a production $A \to [B,C]$ such that B and C each derive an appropriate portion of the final string substring. Testing this condition may involve checking each possibility for splitting the substring into two nonempty substrings. (The range of choices corresponds to the range of possible values of k.)

The Cocke-Younger-Kasami algorithm arises from this recursive algorithm through an application of the technique of dynamic programming. We describe how this is done in the Appendix.

5. Left-to-Right Parsers

In the derivation of the Cocke-Younger-Kasami algorithm, we started with the basic recursive definition of the restricted *derives* relation,

$$\begin{aligned} D(\psi,\pi) \leftarrow \quad & |\psi| = \pi \quad \vee \\ & \psi \to \beta \wedge \beta = [\sigma_1,\ldots,\sigma_n] \wedge \\ & \quad D(\sigma_1,\delta_1) \wedge \cdots \wedge D(\sigma_n,\delta_n) \wedge \\ & \quad \delta_1 \circ \cdots \circ \delta_n = \pi. \end{aligned}$$

Our assumption that the grammar was in Chomsky Normal Form let us simplify the definition to eliminate the ellipses. In the derivations of Earley's algorithm and of the top-down algorithm, we do not make such assumptions about the grammar, and so we cannot eliminate the ellipses by simplification.

Instead, we eliminate the ellipses by committing ourselves to a particular order of computation of the conjunction, say from left to right. The resulting definition of D will thus characterize *leftmost* derivations.

Eliminating the ellipses In the section on notations we showed that there were several equivalent ways to expand a notation like $P_1 \wedge \cdots \wedge P_n$. One of these corresponds to a left-to-right evaluation of the conjunction:

$$P_i \wedge \cdots \wedge P_j \leftarrow \textbf{if } i=j-1 \textbf{ then true else } P_i \wedge (P_{i+1} \wedge \cdots \wedge P_j).$$

(We assume a qualifier restricting the variables i and j to nonnegative integers such that $i \leq j+1$.)

We will use this left-to-right schema to form a specialized expression procedure for the expression

$$\begin{aligned}&[\exists n, \sigma_1, \ldots, \sigma_n, \delta_1, \ldots, \delta_n]\\ &\quad \beta=[\sigma_1, \ldots, \sigma_n] \wedge\\ &\quad D(\sigma_1, \delta_1) \wedge \cdots \wedge D(\sigma_n, \delta_n) \wedge\\ &\quad \delta_1 \circ \cdots \circ \delta_n = \pi,\end{aligned}$$

which occurs in the body of D. (Note that this expression has only two free variables, β and π.)

The manipulations we now perform to obtain the new expression procedure are a bit tedious, but (we believe) they are easily mechanizable.

We start by instantiating the left-to-right schema above to the case where P_i is $D(\sigma_i, \delta_i)$.

$$\begin{aligned}&D(\sigma_i, \delta_i) \wedge \cdots \wedge D(\sigma_j, \delta_j) \leftarrow\\ &\qquad \textbf{if } i=j-1 \textbf{ then true}\\ &\qquad\quad \textbf{else } D(\sigma_i, \delta_i) \wedge (D(\sigma_{i+1}, \delta_{i+1}) \wedge \cdots \wedge D(\sigma_j, \delta_j))\end{aligned}$$

We compose this into an expression involving the strings β and π:

$$\begin{aligned}&\beta=[\sigma_i, \ldots, \sigma_j] \wedge\\ &\quad D(\sigma_i, \delta_i) \wedge \cdots \wedge D(\sigma_j, \delta_j) \wedge\\ &\quad \delta_i \circ \cdots \circ \delta_j = \pi\\ &\leftarrow\\ &\qquad \beta=[\sigma_i, \ldots, \sigma_j] \wedge\\ &\qquad\quad (\textbf{if } i=j-1 \textbf{ then true}\\ &\qquad\qquad \textbf{else } D(\sigma_i, \delta_i) \wedge (D(\sigma_{i+1}, \delta_{i+1}) \wedge \cdots \wedge D(\sigma_j, \delta_j))) \wedge\\ &\qquad\quad \delta_i \circ \cdots \circ \delta_j = \pi\end{aligned}$$

The two additional conjuncts can be distributed into the conditional and the result simplified to obtain

$$\begin{aligned}&\beta=[\sigma_i, \ldots, \sigma_j] \wedge\\ &\quad D(\sigma_i, \delta_i) \wedge \cdots \wedge D(\sigma_j, \delta_j) \wedge\\ &\quad \delta_i \circ \cdots \circ \delta_j = \pi\\ &\leftarrow\\ &\qquad \textbf{if } i=j-1 \textbf{ then } \beta=\Lambda \wedge \pi=\Lambda\\ &\qquad\quad \textbf{else } \beta=[\sigma_i, \ldots, \sigma_j] \wedge\\ &\qquad\qquad D(\sigma_i, \delta_i) \wedge \delta_i \circ \gamma = \pi \wedge\\ &\qquad\qquad D(\sigma_{i+1}, \delta_{i+1}) \wedge \cdots \wedge D(\sigma_j, \delta_j) \wedge\\ &\qquad\qquad \delta_{i+1} \circ \cdots \circ \delta_j = \gamma.\end{aligned}$$

(A new existential variable γ was introduced in the body. Its quantification can be bounded to the set of suffixes of the string π.)

Recall that we are only interested in applying this expression procedure to an expression whose variables are all existential, except those corresponding to β and π. We therefore compose further, adding an existential quantifier for all variables in the name and body except β and π. This will give us additional flexibility in simplifying the body.

$$\begin{aligned}&[\exists i, j, \sigma_i, \ldots, \sigma_j, \delta_i, \ldots, \delta_j]\\ &\quad \beta=[\sigma_i, \ldots, \sigma_j] \wedge\\ &\quad D(\sigma_i, \delta_i) \wedge \cdots \wedge D(\sigma_j, \delta_j) \wedge\\ &\quad \delta_i \circ \cdots \circ \delta_j = \pi\\ &\leftarrow\\ &\qquad [\exists i, j, \sigma_i, \ldots, \sigma_j, \delta_i, \ldots, \delta_j]\\ &\qquad \textbf{if } i=j-1 \textbf{ then } \beta=\Lambda \wedge \pi=\Lambda\\ &\qquad\quad \textbf{else } \beta=[\sigma_i, \ldots, \sigma_j] \wedge\\ &\qquad\qquad D(\sigma_i, \delta_i) \wedge \delta_i \circ \gamma = \pi \wedge\\ &\qquad\qquad D(\sigma_{i+1}, \delta_{i+1}) \wedge \cdots \wedge D(\sigma_j, \delta_j) \wedge\\ &\qquad\qquad \delta_{i+1} \circ \cdots \circ \delta_j = \gamma\end{aligned}$$

We now simplify this definition by renaming and eliminating existential variables. In the name, we eliminate i and j in favor of 1 and n. We do this

by introducing a new set of existential variables, $\rho_1, \ldots, \rho_n$ and constraining them by the conditions

$$n = j - i + 1 \quad \wedge \quad \rho_1 = \sigma_i \quad \wedge \cdots \wedge \quad \rho_n = \sigma_j.$$

Since these conditions are satisfiable, they do not affect the meaning of the definition, and so we can modify the name term to use these new variables in the place of the σ variables, ultimately eliminating the σ variables in the same way that we introduced the ρ variables. The final step in this simplification is to rename to ρ's to σ's

In the body, we first transform the conditional into a disjunction, eliminating the now superfluous test. Second, we use δ for δ_1, $hd(\beta)$ for σ_i, and, as above, we write 1 and n where we used $i + 1$ and j previously. Finally, we bring the quantifier itself inward.

$$\begin{array}{l}
[\exists n, \sigma_1, \ldots, \sigma_n, \delta_1, \ldots, \delta_n] \\
\quad \beta = [\sigma_1, \ldots, \sigma_n] \ \wedge \\
\quad D(\sigma_1, \delta_1) \ \wedge \cdots \wedge D(\sigma_n, \delta_n) \ \wedge \\
\quad \delta_1 \circ \cdots \circ \delta_n = \pi \\
\leftarrow \\
\qquad \beta = \Lambda \ \wedge \ \pi = \Lambda \quad \vee \\
\qquad D(hd(\beta), \delta) \ \wedge \ \delta \circ \gamma = \pi \ \wedge \\
\qquad\quad [\exists n, \sigma_1, \ldots, \sigma_n, \delta_1, \ldots, \delta_n] \\
\qquad\quad (tl(\beta) = [\sigma_1, \ldots, \sigma_n] \ \wedge \\
\qquad\quad D(\sigma_1, \delta_1) \ \wedge \cdots \wedge D(\sigma_n, \delta_n) \ \wedge \\
\qquad\quad \delta_1 \circ \cdots \circ \delta_n = \gamma)
\end{array}$$

(Note that the value of n in the body will now always be one less than the value of n in the name.)

This expression procedure is recursive, so by renaming we will be able to eliminate all uses of ellipses from the program. We first abstract the body of the expression procedure:

$$\begin{array}{l}
[\exists n, \sigma_1, \ldots, \sigma_n, \delta_1, \ldots, \delta_n] \\
\quad \beta = [\sigma_1, \ldots, \sigma_n] \ \wedge \\
\quad D(\sigma_1, \delta_1) \ \wedge \cdots \wedge D(\sigma_n, \delta_n) \ \wedge \\
\quad \delta_1 \circ \cdots \circ \delta_n = \pi \\
\leftarrow \\
\qquad Q(\beta, \pi) \qquad\qquad (*)
\end{array}$$

(The new basic name that results from abstraction has only two parameters, since only the variables β and π are free in the definition.) The definition of Q, after unfolding the call to this new expression procedure, becomes quite simple:

$$\begin{array}{lll}
Q(\beta, \pi) \leftarrow & \beta = \Lambda \ \wedge \ \pi = \Lambda & \vee \\
& D(hd(\beta), \delta) \ \wedge \ \delta \circ \gamma = \pi \ \wedge \ Q(tl(\beta), \gamma).
\end{array}$$

The last remaining call to the expression procedure is in the body of the definition of D. After unfolding this, we can eliminate the expression procedure, and thus the last trace of the ellipses. We are left with the two mutually recursive definitions:

$$\begin{array}{lll}
D(\psi, \pi) \leftarrow & |\psi| = \pi & \vee \quad \psi \rightarrow \beta \ \wedge \ Q(\beta, \pi) \\
Q(\beta, \pi) \leftarrow & \beta = \Lambda \ \wedge \ \pi = \Lambda & \vee \\
& D(hd(\beta), \delta) \ \wedge \ \delta \circ \gamma = \pi \ \wedge \ Q(tl(\beta), \gamma) \\
\underline{\text{Compute}} & D([S], [a_1, \ldots, a_M]) &
\end{array}$$

These definitions have the interesting property that they *both* describe the *derives* relation; $D(\psi, \pi)$ is true when the symbol ψ derives the string π, and $Q(\beta, \pi)$ is true when the string β derives the string π. Thus, the new definition Q describes a more general version of the *derives* relation than does the old definition D.

Elimination of D. Because of this property, we will find in the later derivations that it is easier to manipulate Q than D. We therefore choose to eliminate D in favor of Q, by means of application and simplification.

(An automatic system could have made the same choice: After failing to eliminate Q in favor of D, it would make sense to try eliminating D in favor

of Q before giving up. The latter elimination would have been successful, as we now show.)

We start by unfolding the call to D in the body of Q

$$\begin{aligned} Q'(\beta,\pi) \leftarrow\ & \beta = \Lambda \wedge \pi = \Lambda \ / \\ & (hd'(\beta) = \delta \ / \ hd'(\beta) = a \wedge Q(a,\delta)) \wedge \\ & \delta \circ \gamma = \pi \wedge Q'(tl'(\beta),\gamma) \end{aligned}$$

(The existential variable β in D was renamed to a to prevent a name conflict.) Distributing the disjunction through and simplifying gives

$$\begin{aligned} Q'(\beta,\pi) \leftarrow\ & \beta = \Lambda \wedge \pi = \Lambda \ / \\ & hd'(\beta) = hd(\pi) \wedge Q'(tl'(\beta),tl(\pi)) \ \vee \\ & hd'(\beta) = a \wedge Q'(a,\delta) \wedge Q(tl(\beta),\gamma) \wedge \delta \circ \gamma = \pi. \end{aligned}$$

In the second disjunct, we observed that since δ is a singleton list, we can write $tl(\pi)$ for γ.

It remains to eliminate the call to D in the Compute specification,

$$\text{Compute}\quad D'(S,[a_1,\ldots,a_M])$$

Unfolding the definition of D and eliminating the false first disjunct yields

$$\text{Compute}\quad S = a \wedge Q(a,[a_1,\ldots,a_M]).$$

(The definition of D can now be eliminated.) We can simplify this Compute specification in a rather unusual way. Observe that this specification is an instance of the body of Q in the case that β is $[S]$ and π is $[a_1,\ldots,a_M]$. That is, we can abstract both into a new definition, say for $Q'(\beta,\pi)$. The result will be the program

$$\begin{aligned} Q'(\beta,\pi) \leftarrow\ & Q''(\beta,\pi) \\ Q''(\beta,\pi) \leftarrow\ & \beta = \Lambda \wedge \pi = \Lambda \ / \\ & hd'(\beta) = hd'(\pi) \wedge Q'(tl'(\beta),tl(\pi)) \ \vee \\ & hd'(\beta) = a \wedge Q'(a,\delta) \wedge Q(tl(\beta),\gamma) \wedge \delta \circ \gamma = \pi \\ \text{Compute}\quad & Q''(S,[a_1,\ldots,a_M]) \end{aligned}$$

(Because of our abstraction, this definition of Q' is identical to the previous definition of Q'.) We now unfold all instances of Q and eliminate its definition from the program. We can therefore rename 'Q''' to 'Q',' giving our final program,

$$\begin{aligned} Q'(\beta,\pi) \leftarrow\ & \beta = \Lambda \wedge \pi = \Lambda \ / \\ & hd'(\beta) = hd'(\pi) \wedge Q'(tl'(\beta),tl(\pi)) \ \vee \\ & hd'(\beta) = a \wedge Q(a,\delta) \wedge Q(tl(\beta),\gamma) \wedge \delta \circ \gamma = \pi \\ \text{Compute}\quad & Q'(S,[a_1,\ldots,a_M]) \end{aligned}$$

This defines a *basic left-to-right version of the derives relation*: A string β derives a string π if one of the following three conditions holds: (a) Both strings are empty. (b) the leftmost symbols in the two strings β and π are equal, and the tail of the β derives the tail of π, or (c) there is a production from the leftmost symbol of β to a string that derives some prefix of π, and the remainder of π is derived by the tail of β.

A right-to-left algorithm The above algorithm was developed by using a left-to-right elimination schema. We can develop a right-to-left version of the algorithm by starting instead from a right-to-left notation definition:

$$P_i \wedge \cdots \wedge P_j \leftarrow \text{if } i = j+1 \text{ then true else } (P_i \wedge \cdots \wedge P_{j-1}) \wedge P_j$$

The algorithm that results is a mirror-image of the left-to-right algorithm:

$$\begin{aligned} Q'(\beta,\pi) \leftarrow\ & \beta = \Lambda \wedge \pi = \Lambda \ \vee \\ & Q'(fr(\beta),fr(\pi)) \wedge lt(\beta) = lt(\pi) \ \vee \\ & Q'(fr(\beta),\gamma) \wedge lt(\beta) \rightarrow a \wedge Q(a,\delta) \wedge \gamma \circ \delta = \pi \\ \text{Compute}\quad & Q'(S,[a_1,\ldots,a_M]) \end{aligned}$$

6. Earley's Algorithm

We can develop the recursive Earley's algorithm by specializing the basic right-to-left algorithm developed in the last section. (We use this algorithm rather than the left-to-right version because at the end of the derivation we apply a dynamic programming technique, which will effectively reverse the direction of evaluation, yielding a left-to-right algorithm from this right-to-left definition. See the Appendix for these dynamic programming steps.)

$$\begin{aligned} Q(\beta,\pi) \leftarrow\quad & \beta = \Lambda \wedge \pi = \Lambda \quad \vee \\ & Q(fr(\beta), fr(\pi)) \wedge lt(\beta) = lt(\pi) \quad \vee \\ & Q(fr(\beta),\gamma) \wedge lt(\beta) \rightarrow \alpha \wedge Q(\alpha,\delta) \wedge \gamma \circ \delta = \pi. \end{aligned}$$

Compute $Q([S],[a_1,\ldots,a_M])$

We develop our initial version of Earley's algorithm by performing two steps of specialization on this algorithm.

Specialization: π is a final string substring. The first specialization step is based on a conjecture about the second argument to Q: As in the Cocke-Younger-Kasami derivation, the string π appears always to be a substring of the final string, $[a_1,\ldots,a_M]$. We test this hypothesis by creating an expression procedure for the case in which π is the string $[a_{i+1},\ldots,a_j]$ where $0 \le i \le j \le M$:

$$\begin{aligned} Q(\beta,[a_{i+1},\ldots,a_j]) \leftarrow\quad & \beta = \Lambda \wedge [a_{i+1},\ldots,a_j] = \Lambda \quad \vee \\ & Q(fr(\beta), fr([a_{i+1},\ldots,a_j])) \wedge \\ & \quad lt(\beta) = lt([a_{i+1},\ldots,a_j]) \quad \vee \\ & Q(fr(\beta),\gamma) \wedge lt(\beta) \rightarrow \alpha \wedge \\ & \quad Q(\alpha,\delta) \wedge \gamma \circ \delta = [a_{i+1},\ldots,a_j]. \end{aligned}$$

We can take $\gamma = [a_{i+1},\ldots,a_k]$ for some k, and simplify this to

$$\begin{aligned} Q(\beta,[a_{i+1},\ldots,a_j]) \leftarrow\quad & \beta = \Lambda \wedge i = j \quad \vee \\ & Q(fr(\beta),[a_{i+1},\ldots,a_{j-1}]) \wedge lt(\beta) = a_j \quad \vee \\ & Q(fr(\beta),[a_{i+1},\ldots,a_k]) \wedge lt(\beta) \rightarrow \alpha \wedge \\ & \quad Q(\alpha,[a_{k+1},\ldots,a_j]). \end{aligned}$$

(The quantification of the new existential variable k can be bounded to the interval $i \le k \le j$.) Thus, our specialization is successful, and so we rename to eliminate the explicit substring references. We parameterize $[a_{i+1},\ldots,a_j]$ in terms of i, j, and the final string $[a_1,\ldots,a_M]$. As before, we leave the final string as an implicit global variable, rather than carrying it along as an extra constant argument.

$$\begin{aligned} Q'(\beta,i,j) \leftarrow\quad & \beta = \Lambda \wedge i = j \quad \vee \\ & Q'(fr(\beta),i,j-1) \wedge lt(\beta) = a_j \quad \vee \\ & Q'(fr(\beta),i,k) \wedge lt(\beta) \rightarrow \alpha \wedge Q'(\alpha,k,j) \end{aligned}$$

Compute $Q'([S],0,M)$

Specialization: β is a production prefix. The previous specialization step *restricted the range of the second argument to Q to be a substring* of the final string, thus giving the algorithm a bottom-up flavor. Now, we shall restrict the range of the first argument, adding a top-down element to the program. This specialization step is particularly revealing about the conceptual basis of Earley's algorithm.

The observation we make is this: The program can be modified in such a way that the string β is always the prefix of the right-hand side of some production.

To test this hypothesis, we compose Q' (which we will now call Q, since the original Q was eliminated by the previous specialization step) with an appropriate expression involving the production relation.

$$\begin{aligned} A \rightarrow \beta \circ \gamma \wedge Q(\beta,i,j) \leftarrow\quad & \\ & A \rightarrow \beta \circ \gamma \wedge \beta = \Lambda \wedge i = j \quad \vee \\ & A \rightarrow \beta \circ \gamma \wedge Q(fr(\beta),i,j-1) \wedge lt(\beta) = a_j \quad \vee \\ & A \rightarrow \beta \circ \gamma \wedge Q(fr(\beta),i,k) \wedge lt(\beta) \rightarrow \alpha \wedge Q(\alpha,k,j) \end{aligned}$$

Here we used composition to add the primitive predicate $A \rightarrow \beta \circ \gamma$ to both sides of the definition, distributing it through the disjunction in the body. Our goal is to simplify the body in such a way that all calls to Q have the specialized form.

We start by making several straightforward simplifications based on the properties of strings.

$$\begin{aligned} A \to \beta \circ \gamma \ \wedge\ & Q(\beta, i, j) \leftarrow \\ & A \to \gamma \ \wedge\ \beta = \Lambda \ \wedge\ i = j \quad \vee \\ & A \to fr(\beta) \circ ([a_j] \circ \gamma) \ \wedge\ Q(fr(\beta), i, j-1) \ \wedge\ lt(\beta) = a_j \quad \vee \\ & A \to fr(\beta) \circ ([lt(\beta)] \circ \gamma) \ \wedge\ Q(fr(\beta), i, k) \ \wedge \\ & \quad lt(\beta) \to \alpha \circ \Lambda \ \wedge\ Q(\alpha, k, j). \end{aligned}$$

Compute: $Q([S], 0, M)$

Here, we have twice used the property that $\beta = fr(\beta) \circ [lt(\beta)]$. All of the recursive calls to Q are now in the specialized form. To handle the single call in the Compute step, which is not in the desired form, we make use of the application rule and expand the definition of Q.

$$\begin{aligned} \text{Compute: } [\exists \alpha, k] \quad & [S] = \Lambda \ \wedge\ 0 = M \quad \vee \\ & Q(fr([S]), 0, M-1) \ \wedge\ lt([S]) = a_M \quad \vee \\ & Q(fr([S]), 0, k) \ \wedge\ lt([S]) \to \alpha \ \wedge\ Q(\alpha, k, M). \end{aligned}$$

The first two disjuncts are false, so the specification simplifies to

$$\text{Compute: } [\exists \alpha, k] \quad Q(\Lambda, 0, k) \ \wedge\ S \to \alpha \ \wedge\ Q(\alpha, k, M).$$

Unfolding the call $Q(\Lambda, 0, k)$ and simplifying the result yields

$$\text{Compute: } [\exists \alpha, k] \quad k = 0 \ \wedge\ S \to \alpha \ \wedge\ Q(\alpha, k, M),$$

or

$$\text{Compute: } [\exists \alpha] \quad S \to \alpha \circ \Lambda \ \wedge\ Q(\alpha, 0, M),$$

which is in the specialized form. (Note that in this last application step, we essentially undid the step of abstraction at the end of the derivation of the basic right-to-left algorithm.)

As before, we rename to complete the specialization sequence. The new definition we obtain has five parameters.

$$\begin{aligned} Q'(A, \beta, \gamma, i, j) \leftarrow \quad & A \to \gamma \ \wedge\ \beta = \Lambda \ \wedge\ i = j \quad \vee \\ & Q'(A, fr(\beta), ([a_j] \circ \gamma), i, j-1) \ \wedge\ lt(\beta) = a_j \quad \vee \\ & Q'(A, fr(\beta), ([lt(\beta)] \circ \gamma), i, k) \ \wedge\ Q'(lt(\beta), \alpha, \Lambda, k, j) \end{aligned}$$

Compute: $[\exists \alpha] \quad Q'(S, \alpha, \Lambda, 0, M)$

We now have an approximation to a recursive implementation of Earley's algorithm. Observe that we can interpret the five parameters of Q' in terms of Earley's state set notation ([Earley68]):

$$\text{If} \quad [A \to \beta . \gamma, i] \in I_j \quad \text{then} \quad Q'(A, \beta, \gamma, i, j).$$

In addition, the structure of the algorithm (with the exception of the first disjunct) corresponds exactly with the zero look-ahead version of Earley's algorithm. The first disjunct in this version combines Earley's initiation and *predictor* steps; the second disjunct corresponds to his *scanner* step, and the third disjunct corresponds to the *completer* step.

The actual implementation of Earley's algorithm involves the construction of a tableau representing a dynamic programming array for this recursive algorithm Q'. (We describe in the Appendix how the first disjunct is refined into the initiation and *predictor* steps, and how the dynamic programming transformation is performed.)

7. The Top-Down Algorithm

In this final derivation, we start with the basic left-to-right algorithm.

$$\begin{aligned} Q(\beta, \pi) \leftarrow \quad & \beta = \Lambda \ \wedge\ \pi = \Lambda \quad \vee \\ & hd(\beta) = hd(\pi) \ \wedge\ Q(tl(\beta), tl(\pi)) \quad \vee \\ & hd(\beta) \to \alpha \ \wedge\ Q(\alpha, \delta) \ \wedge\ Q(tl(\beta), \gamma) \ \wedge\ \delta \circ \gamma = \pi. \end{aligned}$$

Compute: $Q([S], [a_1, \ldots, a_M])$.

To obtain the top-down (and shift-reduce) algorithms, we transform this algorithm in such a way that the recursion becomes linear. This will enable us to develop an efficient iterative implementation that avoids the need for the dynamic programming array required by the two previous parsing algorithms. We do this by proving a property of Q that will enable us to combine the two recursive calls in the last disjunct.

Combining two recursions. In the first step of this derivation we make a transformation that may affect the termination properties of Q. To make such a change, we cannot use the transformation rules (since they preserve termination properties), and so instead we must rely on the Program Substitution Theorem, which guarantees only that *if* the new program terminates, it will have results equivalent to the original program. Our burden, then, will be to prove termination properties of the new program after completion of the transformation. While the algorithm (like the initial *derives* program) will not terminate for all grammars, it is possible to establish sufficient conditions for termination.

The transformation we make involves the two recursive calls to Q in the last disjunct. We can prove by induction the following property of Q, which corresponds to a well-known property of the *derives* relation:

For any α_1, α_2, and π,

$$[\exists\, \sigma_1, \sigma_2] \quad Q(\alpha_1, \sigma_1) \wedge Q(\alpha_2, \sigma_2) \wedge \sigma_1 \circ \sigma_2 = \pi$$

if and only if

$$Q(\alpha_1 \circ \alpha_2, \pi)$$

(This property follows directly from the expression procedure definition that is marked '(*)' in Section 5.)

Thus, by (an appropriate extension to) the Program Substitution Theorem of Chapter 2, we replace the term

$$[\exists\, \delta, \gamma] \quad Q(\alpha, \delta) \wedge Q(tl(\beta), \gamma) \wedge \delta \circ \gamma = \pi$$

in the definition of Q by the term $Q(\alpha \circ tl(\beta), \pi)$. The resulting definition is

$$\begin{aligned} Q(\beta, \pi) \leftarrow \quad & \beta = \Lambda \wedge \pi = \Lambda \quad \vee \\ & hd(\beta) = hd(\pi) \wedge Q(tl(\beta), tl(\pi)) \quad \vee \\ & hd(\beta) \rightarrow \alpha \wedge Q(\alpha \circ tl(\beta), \pi) \end{aligned}$$

Compute: $Q([S], [a_1, \ldots, a_M])$.

Specialization: π is a suffix of the final string. We now specialize this algorithm according to the hypothesis that the second argument π of Q is always a suffix of the final string, $[a_1, \ldots, a_M]$.

We compose to form an expression procedure for $Q(\beta, [a_i, \ldots, a_M])$:

$$\begin{aligned} Q(\beta, [a_i, \ldots, a_M]) \leftarrow \quad & \beta = \Lambda \wedge [a_i, \ldots, a_M] = \Lambda \quad \vee \\ & hd(\beta) = hd([a_i, \ldots, a_M]) \wedge Q(tl(\beta), tl([a_i, \ldots, a_M])) \vee \\ & hd(\beta) \rightarrow \alpha \wedge Q(\alpha \circ tl(\beta), [a_i, \ldots, a_M]). \end{aligned}$$

Simplification yields

$$\begin{aligned} Q(\beta, [a_i, \ldots, a_M]) \leftarrow \quad & \beta = \Lambda \wedge i = M+1 \quad \vee \\ & hd(\beta) = a_i \wedge Q(tl(\beta), [a_{i+1}, \ldots, a_M]) \quad \vee \\ & hd(\beta) \rightarrow \alpha \wedge Q(\alpha \circ tl(\beta), [a_i, \ldots, a_M]), \end{aligned}$$

so our specialization is successful.

We rename to eliminate the expression procedure: **Abstraction of the** body and application of the three calls to Q yields

$$\begin{aligned} Q'(\beta, i) \leftarrow \quad & \beta = \Lambda \wedge i = M+1 \quad \vee \\ & hd(\beta) = a_i \wedge Q'(tl(\beta), i+1) \quad \vee \\ & hd(\beta) \rightarrow \alpha \wedge Q'(\alpha \circ tl(\beta), i) \end{aligned}$$

Compute: $Q'([S], 1)$.

(As before, we avoid an additional constant argument by regarding the final string $[a_1, \ldots, a_M]$ as implicit.)

This algorithm is a simple recursive top-down parser. Here, β represents the stack, and i the current position in the input string. Initially, the stack has the only the start symbol S on it, and the current input pointer is positioned at the first symbol in the input string. The three disjuncts correspond to the three choices we have at any time during a top-down parse: (a) If the stack is empty and the entire input string has been scanned, then the parse is successful. (b) If the top symbol on the stack is equivalent to the current symbol in the input string, then that symbol can be popped off the stack and the input pointer advanced. (c) Finally (if the top stack symbol is

nonterminal), choose a production whose left-hand side matches the top of the stack and replace that symbol on the stack by the right-hand side of the production, leaving the current input pointer unchanged.

Observe that because of the last disjunct, this algorithm is nondeterministic: Any production with the correct left-hand side could be chosen.

Recall that the first step of this derivation involved use of the Program Substitution Theorem, rather than use of the equivalence-preserving transformations. Therefore, we must explicitly establish the termination properties of our top-down parsing program

We first observe that the program *does not always terminate*. For example, if the grammar contains the production $S \rightarrow S$ (or, more generally, if the grammar is left-recursive) then a loop can be generated through the last disjunct. Therefore, we will need to restrict the grammar in some way to guarantee termination. The most natural set of restrictions involves the notion of *loop-free*-ness

A grammar is *loop-free* if for any distinct nonterminal symbols A and B such that $A \stackrel{+}{\Rightarrow} B$, then it is not the case that $B \stackrel{+}{\Rightarrow} A$.

We can specialize Q' to the case in which the grammar described by P is loop-free and contains no erasing rules. In this case, as long as i remains constant, the stack β cannot diminish in size. The key step of this specialization is an induction proof that if the size of the stack ever exceeds $M - i + 1$, then the parse will fail (i.e., Q' will be **false**). We therefore add a test for this possibility in the last disjunct. We can then prove, using a well-founded ordering based on i and the size of the stack, that evaluation of Q' always terminates for this specialized case.

Chapter 6

Conclusions

In this final chapter, we assess the potential impact of this work, and suggest areas for future investigation. It is likely that program transformation *research will have a significant impact on our methodology of programming*, and, more directly, on the programming tools we use and our methods for developing and explaining complex algorithms.

In the next section, we explore the connection between program derivation and programming methodology.

In the section following, we consider how the present work can be extended in order to make it useful in the construction of programming tools, and in the exposition of algorithms.

1. A Note on Programming Methodology

The implementation of a particular methodology of programming usually involves the creation of tools designed to encourage an effective and efficient style of programming

In the past this has been accomplished strictly through the means of programming language design. The idea is to design programming languages so that their structure reflects the way we organize our own problem solving, rather than to base them only on underlying machine operations. Thus, a programming language design must balance between the often conflicting goals of conceptual clarity and potential efficiency of object code. Enforcement of a particular methodology may be possible only at some cost of potential program efficiency.

There is now an increasing trend towards the development of program manipulation tools that, because of their focus on the explicit representation of the conceptual elements of the programming *process*, place less emphasis on the design of the actual target language. These tools are designed to help the programmer work systematically from specification to program. They enforce a particular methodology by limiting his choices along the way, and by requiring him to describe his goals within the structure of a particular language.

It may be that neither of these two approaches to the implementation of a programming methodology is the best one. By tacitly interpreting conceptual structuring as the simple enforcement of rules and restrictions, both of these approaches can severely limit the creative thinking of the programmer.

The first approach, constraining a programming language design to facilitate understanding and analysis of programs, may in addition critically impair facility of coding and ultimate program efficiency. It seems preferable in the process of programming to keep separate these dual purposes of communicating and analyzing algorithms on the one hand, and implementing correct efficient programs on the other.

The second approach, the use of structure-inducing programming tools, may have the side effect of preventing the attention of a programmer from shifting freely from one aspect of his program to another. To allow the programmer's intuition to be his guide, programming tools must be designed for maximum flexibility.

We propose that the goal of the real process of program construction must be the development not just of a target program, but the development of an entire program derivation. This derivation, which is itself a highly structured object, is intended to reveal the conceptual structure behind what may be an otherwise perplexing program. Thus, the enforcement of structuring occurs only in the derivation itself – not in the target program or in the process of developing the program derivation. This enables us to avoid the mistake of identifying program construction in time with program derivation in concept. (See [Bates79], [Knuth74b], and [McKeeman75] for further discussion.)

In spite of such elegant exceptions as [Dijkstra76], the method of communication of a program still generally consists of a single program in the target language, and so much of the activity of programming is geared towards the production of "self-documenting" programs.

We can see that the communication and documentation of programs is facilitated through the use of program derivations, since a program derivation must by its nature reveal (a) the abstract program elements and their representations, (b) how efficiency is improved through optimization, and (c) what logical facts are used to make these steps. Of course the quality of the explication given a program by such a program derivation sequence depends both on the quality of the design of the language used to represent program derivations and on the level of understanding the programmer has of his program, as revealed in this language.

The comparison of program derivation sequences for a group of algorithms can teach us much about the relations between algorithms. However, such developmental taxonomies, such as those of sorting algorithms by Darlington and by Green and Barstow, are not easily produced — the sorting algorithms were known and in common use long before the relations between them were perceived. The act of producing the complete derivation requires the programmer to express his intuitions within the framework of an external language; thus it will certainly be harder to produce a complete program derivation than to write just a target program. This additional effort is very often justified, however, by the added understanding resulting from the use of program derivation, and by the practical advantages lent to the production and modification of correct programs.

In the process of program derivation, we will be concerned with both the development of new programs and the modification of existing programs. It is advantageous to modify an old program rather than to develop an entirely new program when we perceive that the developments of the old and new programs would have much in common; that is, that there would be significant sharing in the program derivations, even if the resulting target programs have little in common. The use of explicit program derivations clearly enhances our ability to modify programs, since the sharing is made explicit.

Thus, when we refer to the systematic development of programs, we mean the development of systematic program derivations. Structure is imposed not on the shape of the final target program, but on the derivation sequence that leads us there. Principles of programming are enforced not by constraining the target language of programs, nor by constraining the programming tools, but

instead by restricting the structure of program derivation sequences. (Thus, when we speak of program transformations, we mean transitions from step to step in program derivation sequences.)

A successful programming methodology must therefore be based on a theory of the *development* of programs, embodying the basic principles of programming, from which are developed practical methods for the representation and manipulation of programs derivations.

2. Future Directions

Because of its broad scope, this thesis has raised more problems than it has presented solutions. We suggest here several topics for further research. These topics naturally divide into the three application areas mentioned in Chapter 4: The development of automatic systems and programming tools, very high-level languages, and the formal development of algorithms. In addition, we list several theoretical questions that are of potential interest.

Automatic systems. We first discuss considerations related to the implementation of a general purpose program derivation system based on expression procedures and specialization. Such a system would include an implementation of the expression procedure language, procedures for applying the transformation rules, and a heuristic facility for directing the derivations.

The heuristic aspect of the specialization process needs to be investigated in greater depth. While we note that many existing program transformation heuristics, such as the generalization heuristic of Manna and Waldinger and the UF heuristics of Feather, can be used as specialization heuristics, additional investigation is necessary to determine specializations tactics that are successful in specific problem domains. For example, the experience of the parser derivations leads us to believe that gradual specialization yields the most favorable results.

It appears that for the purposes of experimentation, and probably in practice as well, an interactive facility would be most desirable. To develop a successful implementation, considerable effort must be put into the design of the command language — the programming *meta-language*. This language

must provide (as detailed in the last section) a flexible means for specifiying the construction of program derivations.

We note that such an implementation would probably require only a modest theorem proving capability: A glance at the parser derivations reveals how mild the theorem proving requirements are for even the most complicated derivations. On the other hand, a sophisticated heuristic facility is required to help direct the specialization process.

Besides the development of a general semi-automatic program development system there are potential applications of our techniques to the development of a specific programming tool for the specialization of programs. Such a tool would have applications both to the development of compilers and more generally as part of a programming environment. An example of a practical use for such a facility is the automatic specialization of a program using abstract data types to a particular set of representations for those types. (The role of a specialization mechanism in such a facility is suggested in [Wegbreit73].)

Very high-level languages. Program transformation techniques have obvious importance in the implementation of very high-level languages (such as SETL and APL), in which some significant amount of optimization must be done before a program reaches an acceptable level of efficiency. The techniques we describe for notations have potential application here. In addition, very high-level languages provide a good setting for the development of program transformation systems, since they enable direct evaluation of, and thus experimentation with, the initial "descriptive" programs in derivations. (See, e.g., [Bauer78] and [Dewar80].)

The formal derivation of algorithms. The formal derivation of a complex algorithm can provide significant insight both into the algorithm itself and into its relation with other algorithms. Such examples, in addition, enable us to refine our derivation techniques to the point where they may be useful in the development of new algorithms.

For example, it is clear that our parser derivations can be extended in various ways, both by refining the algorithms already developed and by developing new algorithms. It would be an interesting exercise to continue

the derivation of Earley's algorithm to the point of developing specific data structures for the representation of the tableau. Another interesting exercise would be the development of the more refined top-down and bottom-up algorithms such as LL and LR, which involve the synthesis of auxiliary programs at the time the grammar is specified.

More generally, it would be of considerable interest to describe within the framework of a program transformation formalism the high level transformations in common use by algorithm designers, such as path compression, divide-and-conquer, and depth-first search.

Technical issues. The technical development in this thesis can be extended in several ways: For example, the composition rule defined in Chapter 2 can be generalized so that the restrictions on the form of the containing term t can be relaxed. It can also be extended so that under certain conditions *any* pair of equivalent terms can be substituted on the left and right hand sides, rather than requiring a definition name on the left and its corresponding body on the right. (This extended rule is based on the notion of lexicographic ordering.) Another possible extension is to the notion of simplification transformation, in which we can introduce rules for "nonprogressive" properties of derivative symbols, such as commutativity.

Finally, we note that it would be of interest to develop more rigorously the relationship between relational programs and dynamic programming as sketched in the Appendix.

Theoretical issues. A variety of interesting theoretical questions are raised by our approach to program transformation. For example: Is there a systematic way to assess the complexity improvement to an algorithm that results from application of a particular program transformation rule? Is there a notion of completeness for program transformation systems such as ours? Are there interesting restrictions of our system that have such a property?

Appendix

A Note on Dynamic Programming

In the previous chapters we have considered only the manipulation of programs in purely applicative programming languages. To develop practical implementations of these programs, however, it is often necessary to make use of imperative features such as assignment and explicit data structures.

In this appendix, we speculate on how this may be done for several of the parsing algorithms derived in Chapter 5. Our goal here is merely to convince the reader of the feasibility of the techniques we suggest, not to develop them in full detail; therefore, the material in this appendix is presented in an informal style, with many technical issues left unaddressed.

1. Linear Recursions

Many recursive programs are of the simple form

$$f(x) \leftarrow \textbf{if } p(x) \textbf{ then } g(x) \textbf{ else } f(h(x)).$$

These programs are said to be *linear* or *iterative* or *tail recursive*, since we can directly transform them into simple loops in an imperative language. For example, the list reversal program

$$rev(x) \leftarrow rev2(x, \Lambda)$$
$$rev2(u, v) \leftarrow \textbf{if } u = \Lambda \textbf{ then } v \textbf{ else } rev2(tl(u), cons(hd(u), v)),$$

developed in Chapter 1 is iterative, and it can be systematically transformed into the imperative program

```
rev(x):  u, v := x, Λ
rev2:    if u = Λ then exit(v)
         u, v := tl(u), cons(hd(u), v)
         goto rev2.
```

(Both assignments are simultaneous assignments.)

As another example, the square root program developed in Chapter 3,

$$s(z) \leftarrow r(0, 1, 3, z)$$
$$r(i, m, n, z) \leftarrow \textbf{if } z < m \textbf{ then } i \textbf{ else } r(i + 1, m + n, n + 2, z),$$

can be transformed into the simple iterative program

```
s(x):    i, m, n := 0, 1, 3
r:       if x < m then exit(i)
         i := i + 1
         m := m + n
         n := n + 2
         goto r.
```

(We have removed two redundant assignments $x := x$.)

This technique of transforming linear recursive programs into simple loops is well known to authors of LISP compilers. (See, e.g., [Steele76].)

2. Nonlinear Programs

Unfortunately, this technique does not extend easily to programs that are not linearly recursive, such as the Fibonacci function,

$$f(x) \leftarrow \textbf{if } x \leq 1 \textbf{ then } x \textbf{ else } f(x - 1) + f(x - 2).$$

An iterative implementation of this program requires some form of explicit data structure for maintaining the context information that is ordinarily implicit in a recursive evaluation. The problem of converting recursive programs into iterative programs has been investigated extensively. It is clear that, except for the simplest cases, such as the linear programs just discussed, data structures need to be introduced in the conversion process. (See Chapter 4 for further discussion.)

Dynamic programming. There is often a high degree of redundancy among the recursive calls in the execution of a program. For example, the computation of $f(4)$ using the definition above requires us to compute $f(1)$ three times. The technique of *dynamic programming* offers a way out of this. In essence, the idea is to compute the values of all the recursive calls in advance, and then determine the function value. We do this by transforming a recursive program into an imperative program that builds up a structured table of values. The difficulty in applying this algorithm improvement technique is in determining (a) the best data structure to use for saving the values, (b) which values need to be saved, and (c) in what order they must be computed. (Dynamic programming is discussed in [Aho74], [Cohen79], and [Bird77b].)

We can thus think of dynamic programming as having the effect of *inverting* the order of evaluation, since it builds from inside out, rather than from outside in, as is the usual case.

Dynamic programming is also similar to Michie's notion of *memo-functions*, in which a function is modified to save in a table the results of all previous computations. Thus, if it is ever called with a set of arguments that has been seen before, it will use the stored value, rather than computing the function value again. Memo-functions are somewhat difficult to implement efficiently, since the values must be stored in a temporal order determined by the recursive evaluator; with dynamic programming, the programmer decides in advance how values will be stored.

In the case of the Fibonacci function, rather than recursively computing values of $f(i)$ for descending values of i, we start by computing $f(0)$ and building up a table of values until the desired value of i has been reached. That is, after $f(i-1)$ and $f(i-2)$ have been computed for some i, the value of $f(i)$ would be appended to the table, and so on, until the desired value is reached. (Note that we need to invert the subtraction function to perform the transformation.) Thus, we compute $f(i)$ in linear time and linear space, a significant improvement over the exponential time and space required by the pre-dynamic programming version.

Dynamic programming for relational programs. In the case of relational programs, we take a slightly different approach to dynamic programming, based on the analogy between evaluation of recursively defined relations and

ordinary backward inference in theorem proving. Consider again the relational definition of the Fibonacci function developed in Chapter 5.

$$F(i,n) \leftarrow \quad i \leq 1 \wedge i = n \quad \vee$$
$$F(i-1,u) \wedge F(i-2,v) \wedge u+v=n$$

Compute: $F(J,N)$

We can view the two recursive calls as *subgoals* that may be generated in the course of an evaluation of $F(i,n)$. The evaluator thus takes on the appearance of a backward-inference theorem-proving program. We can then interpret dynamic programming as the conversion of a backward relational program to use a *forward* inference evaluator. That is, rather than generating subgoals (i.e., recursive calls) whose truth values we are trying to establish, we instead generate a data base of assertions whose values are known to be true. This data base corresponds to the table of values kept in dynamic programming, and the process of filling the table corresponds to the computation of the deductive closure of a relation.

We call this process of converting a backward program into a forward one the *inversion* process.

Before we can develop a forward version of the relational definition of F above, we need to ensure that certain conditions are met. In general, inverting a program may require some modification to that program to ensure that evaluations of the forward version of that program will terminate. This usually entails strengthening the logical condition represented by the definition body, either by simplification or specialization.

In our Fibonacci example, we see that a forward evaluation will not terminate since $F(i,n)$ holds for infinitely many values of i and n. We can, however, restrict the relation to hold for only finitely many values by taking the Compute specification into account. Using a step of specialization, we can add the qualifier $(i \leq J)$ to the definition of F, and thus legitimately add an extra test for this to the second disjunct. After renaming, we have the program

$$F(i,n)\ (i \leq J) \leftarrow \quad i \leq 1 \wedge i = n \quad \vee$$
$$F(i-1,u) \wedge i \leq J \wedge F(i-2,v) \wedge u+v=n$$

Compute: $F(J,N)$.

Observe now that for a given J, F holds for only finitely many pairs of nonnegative i and n.

We *invert* F by writing it as a set of implications, each corresponding to a disjunct in the initial backward program.

$$i \leq 1 \wedge i = n \quad \supset \quad F(i,n)$$
$$F(i-1,u) \wedge i \leq J \wedge F(i-2,v) \wedge u+v=n \quad \supset \quad F(i,n)$$

Compute: $F(J,N)$

Straightforward manipulation of primitive functions yields

$$i \leq 1 \quad \supset \quad F(i,i)$$
$$F(j,u) \wedge j < J \wedge F(j-1,v) \quad \supset \quad F(j+1,u+v)$$

Compute: $F(J,N)$.

This program will be evaluated essentially by performing a closure operation over an initially empty data base. When the data base is empty, only the first implication can be "applied," adding all assertions of the form $F(i,i)$ for all $i \leq 1$; that is, adding $F(0,0)$ and $F(1,1)$. After these two assertions have been added, the second implication is triggered, adding the new assertion $F(2,1)$. This implication then can be applied again adding $F(3,2)$, and so on until j becomes greater than or equal to J, at which point no more assertions can be added and the data base contains the deductive closure of the forward program. The data base is then tested for instances of the Compute specification. If there are any, then the value of the program is true; otherwise it is false.

The next step in the derivation of a forward program could be the development of an efficient data structure and order of computation for the deductive closure. For the Fibonacci program, the natural implementation is to store the values of n in an array indexed on j (after showing by induction that for each j there is exactly one n for which $F(j,n)$ holds).

Dynamic programming and termination. One additional note on our use of dynamic programming: In certain cases, dynamic programming can affect the termination properties of the definitions being transformed. The restrictions we specified above guarantee that an inverted program will terminate

whenever ordinary (backward) evaluation terminates. It is possible, however, that the inverted program will terminate in cases where the backward version does not. Consider, for example, the program $H(x,z)$ for computing the transitive reflexive closure of a relation h:

$$\begin{aligned} H(x,z) \leftarrow \quad & x = z \quad \vee \\ & h(x,z) \quad \vee \\ & H(x,y) \wedge H(y,z) \end{aligned}$$

Compute: $H(X,Z)$.

Here the variables all range over a finite domain and h is an arbitrary relation. This program has nonterminating paths, while the inverted version of this program always terminates.

$$\begin{aligned} \textbf{true} \quad & \supset \quad H(x,z) \\ h(x,z) \quad & \supset \quad H(x,z) \\ H(x,y) \wedge H(y,z) \quad & \supset \quad H(x,z) \end{aligned}$$

Compute: $H(X,Z)$

This forward program terminates because there are only a finite number of possible assertions, and so there can be only finitely many forward inference steps since each step adds a new assertion. The backward program, on the other hand, can generate the same subgoal repeatedly because of the first and last disjuncts. (We can see that it is this property of the inversion operation that enables Earley's algorithm, which is a forward algorithm, to handle loops in grammars, whereas looping derivations must be explicitly prohibited from the top-down algorithm, which is a backward algorithm.)

A footnote on denotational semantics. In the discussion above we have explained the differences in meaning between backward and forward programs in terms of evaluator models. We can also describe the difference using concepts from the fixed point theory of programs. In the case of backward relational programs, our domain of denotations is simply the domain of truth values extended by the special *bottom* value ω, where $\omega \sqsubseteq \mathbf{T}$ and $\omega \sqsubseteq \mathbf{F}$. The conventional two-valued definitions of the propositional connectives can be extended in several ways to include this third special value. Some examples

of possible specifications are

V_1	F	T	ω
F	F	T	ω
T	T	T	ω
ω	ω	ω	ω

V_2	F	T	ω
F	F	T	ω
T	T	T	T
ω	ω	ω	ω

V_3	F	T	ω
F	F	T	ω
T	T	T	T
ω	ω	T	ω

The first of these corresponds to the unordered sequential disjunction operator described in Section 5.1; the second is ordered sequential disjunction, and the last is parallel disjunction. For example, if $\vee$ is the ordered sequential disjunction operation and B is primitive, then the least fixed point of the propositional program for A,

$$A \leftarrow A \vee B,$$

is $A = \omega$. This reflects the fact that backward evaluation of this program for A would loop.

For forward programs, we can use a similar approach to semantics. The only difference is that the domain is structured differently: Recall that the least fixed-point approach is based on a notion of approximation. In the backward model, the looping program denoting ω is the initial approximation to all programs. This is reflected in the two inequalities $\omega \sqsubseteq \mathbf{T}$ and $\omega \sqsubseteq \mathbf{F}$.

In the forward model, the state of the data base reflects the meaning of a program. Our convention is that if the desired assertion is in the data base when the closure operation is complete, then it is true; otherwise it is false. Thus the empty data base, which is our initial approximation, represents the situation in which every assertion is false. The natural domain structure corresponding to this thus has no *bottom* value, and requires that $\mathbf{F} \sqsubseteq \mathbf{T}$.

In this domain then, the least fixed point solution of $A \leftarrow A \vee B$, where B is primitive, is $A = B$. (Note that we only need the two-valued *propositional* operators for this forward domain.) This example illustrates the difference in termination characteristics between backward and forward versions of the same program: "Short circuits" — exact instances of a name in the body of its own definition — cause backward programs to loop, but may have no effect in forward programs. (This example also explains our convention in Chapter 5 regarding errors, that for every relation P, $P(\ldots, \textbf{error}, \ldots) = \textbf{false}$.)

3. The Cocke-Younger-Kasami Algorithm

We do not develop a forward version of the Cocke-Younger-Kasami algorithm here, but rather outline how it is done. Observe first that the program

$$D(A,i,\ell) \leftarrow \quad A \to a_i \wedge \ell{=}1 \quad \vee$$
$$A \to [B,C] \wedge D(B,i,k) \wedge D(C,i{+}k,\ell{-}k) \wedge 0{<}k{<}\ell$$

Compute: $D(S,1,M)$

satisfies the conditions for the dynamic programming transformation. Since this program always terminates, a forward version will also always terminate.

The result of the transformation is that we obtain an $O(M^3)$ parsing algorithm from the exponential recursive algorithm above. One successful dynamic programming implementation is to build up an matrix of sets of nonterminal symbols, indexed on i and ℓ [Aho74]. We can construct this matrix in M^3 steps using three nested loops: The outermost loop advances ℓ from 1 to M; the middle loop advances i from 1 to $M-\ell+1$, and the inner loop advances k from 1 to $\ell-1$. Once the matrix has been constructed, the computation step consists simply of testing whether the start symbol S is in the matrix at the position indexed by $i=1$ and $\ell=M$.

4. Earley's Algorithm

Before inverting a relational program we generally try to reduce through transformations the size of the set of values for which the relation holds, but without affecting the value of the result specified in the Compute specification. This has the effect of reducing the size of the data base generated by the forward version of the program. In the case of the recursive Earley's program,

$$Q'(A,\beta,\gamma,i,j) \leftarrow \quad A \to \gamma \wedge \beta{=}\Lambda \wedge i{=}j \quad \vee$$
$$Q'(A, fr(\beta), ([a_j]\circ\gamma), i, j{-}1) \wedge lt(\beta){=}a_j \quad \vee$$
$$Q'(A, fr(\beta), ([lt(\beta)]\circ\gamma), i, k) \wedge Q'(lt(\beta),\alpha,\Lambda,k,j)$$

Compute: $Q'(S,\alpha,\Lambda,0,M)$,

we attempt to restrict the size of the set of quintuples (A,β,γ,i,j), for which $Q'(A,\beta,\gamma,i,j)$ holds. As in the *fib* example, we do this by means of specialization.

We observe that in an inverted version of this algorithm, a large number of assertions will be made to the data base as a result of the first disjunct: The number of assertions will equal the product of the number of productions in the grammar and the length of the input string. We therefore consider various specialization hypotheses that may allow for a reduction in the size of this set.

The hypothesis we use is this: We are only interested in quintuples in which either A is the start symbol and we are at the beginning of the input string (i.e., $i=0$), or there is some quintuple already in the data base whose third element has A as a prefix and whose current location pointer (the fifth element) is i. That is, our hypothesis is

$$i{=}0 \wedge A{=}S \quad \vee \quad Q'(e_1,e_2,[A]\circ e_3,e_4,i),$$

where e_i are existential variables.

The actual specialization process is omitted here, except to note that special precautions must be taken because of the unusual circumstance that the specialization hypothesis contains an instance of the relation we are specializing. The result of specialization is that our hypothesis is true throughout the definition, and we can simplify accordingly. We obtain

$$Q'(A,\beta,\gamma,i,j) \leftarrow \quad (i{=}0 \wedge A{=}S \quad \vee \quad Q'(e_1,e_2,[A]\circ e_3,e_4,i))$$
$$A \to \gamma \wedge \beta{=}\Lambda \wedge i{=}j \quad \vee$$
$$Q'(A, fr(\beta), ([a_j]\circ\gamma), i, j{-}1) \wedge lt(\beta){=}a_j \quad \vee$$
$$Q'(A, fr(\beta), ([lt(\beta)]\circ\gamma), i, k) \wedge Q'(lt(\beta),\alpha,\Lambda,k,j)$$

Compute: $Q'(S,\alpha,\Lambda,0,M)$.

We can thus add the qualifier conditions to the program as explicit tests.

$$Q'(A,\beta,\gamma,i,j) \leftarrow \quad S \to \gamma \wedge A{=}S \wedge \beta{=}\Lambda \wedge i{=}j{=}0 \quad \vee$$
$$A \to \gamma \wedge \beta{=}\Lambda \wedge i{=}j \wedge Q'(e_1,e_2,[A]\circ e_3,e_4,i) \quad \vee$$
$$Q'(A, fr(\beta), ([a_j]\circ\gamma), i, j{-}1) \wedge lt(\beta){=}a_j \quad \vee$$
$$Q'(A, fr(\beta), ([lt(\beta)]\circ\gamma), i, k) \wedge Q'(lt(\beta),\alpha,\Lambda,k,j)$$

Compute: $Q'(S,\alpha,\Lambda,0,M)$.

We are now ready to invert the program. We obtain four implications, corresponding to the four disjuncts.

$$S \to \gamma \wedge A{=}S \wedge \beta{=}\Lambda \wedge i{=}j{=}0 \supset Q'(A,\beta,\gamma,i,j)$$
$$A \to \gamma \wedge \beta{=}\Lambda \wedge i{=}j \wedge Q'(e_1,e_2,[A]\circ e_3,e_4,i) \supset Q'(A,\beta,\gamma,i,j)$$
$$Q'(A,fr(\beta),([a_j]\circ\gamma),i,j{-}1) \wedge lt(\beta){=}a_j \supset Q'(A,\beta,\gamma,i,j)$$
$$Q'(A,fr(\beta),([lt(\beta)]\circ\gamma),i,k) \wedge Q'(lt(\beta),\alpha,\Lambda,k,j) \supset Q'(A,\beta,\gamma,i,j)$$
Compute: $Q'(S,\alpha,\Lambda,0,M)$.

This is not practically executable by a forward evaluator until we invert the various primitive functions involving integers and strings. This inversion *process uses very simple properties*, replacing implications of the form

$$f(i-1) \supset f(i) \quad \text{by} \quad f(i) \supset f(i+1),$$

and

$$g(fr(\alpha),lt(\alpha)) \supset h(\alpha) \quad \text{by} \quad g(\beta,\psi) \supset h(\beta\circ[\psi]),$$

and finally

$$h(\beta\circ[\psi]) \supset g(\beta,\psi) \quad \text{by} \quad h(\alpha) \supset g(fr(\alpha),lt(\alpha)).$$

Applying transformations such as these to our forward program, we obtain,

$$S \to \gamma \supset Q'(S,\Lambda,\gamma,0,0)$$
$$A \to \gamma \wedge Q'(e_1,e_2,[A]\circ e_3,e_4,i) \supset Q'(A,\Lambda,\gamma,i,i)$$
$$Q'(A,\alpha,([a_j]\circ\gamma),i,j) \supset Q'(A,\alpha\circ[a_{j+1}],\gamma,i,j{+}1)$$
$$Q'(A,\delta,([C]\circ\gamma),i,k) \wedge Q'(C,\alpha,\Lambda,k,j) \supset Q'(A,\delta\circ[C],\gamma,i,j)$$
Compute: $Q'(S,\alpha,\Lambda,0,M)$.

This is Earley's algorithm. In this implementation, $Q'(A,\alpha,\beta,i,j)$ will be asserted to the data base if and only if, in Earley's notation, the quadruple $[A \to \alpha.\beta,i]$ is added to the state set I_j. By examining the complete derivation, we can see that the quintuple $Q'(A,\alpha,\beta,i,j)$ is true if (a) there is a production $A \to \alpha\circ\beta$, and (b) $\alpha \stackrel{*}{\Rightarrow} [a_{i+1},\ldots,a_j]$. The algorithm has the important property that the set of quintuples already asserted is used to determine the actions of the parser: If $Q'(A,\alpha,\beta,i,j)$ is true then we are seeking to find all values of k such that $\beta \stackrel{*}{\Rightarrow} [a_{j+1}\ldots,a_k]$. Thus, we can think of a quintuple as being both an assertion, in that it describes a partially completed parse, and as a goal, in that it is used to determine the next steps taken in the parsing process. It is this combination of bottom-up and top-down that makes the algorithm an efficient one.

As we observed in Chapter 5, Earley's algorithm breaks into four basic steps. The first step is the initialization step, corresponding to the first implication in the definition above, in which we set up the initial goals for the parse. The second implication corresponds to the *predictor* step, which sets up new goals based on partial parse descriptions in the data base. The third implication corresponds to the *scanner*, which satisfies goals involving the next symbol in the final string. The fourth and final implication, corresponding to the *completer*, satisfies goals involving nonterminal symbols.

Further development of this algorithm would involve ordering the computation, and introducing specific data structures for the tableau / data base.

5. The Top-Down Algorithm

The backward left-to-right top-down algorithm we developed in Chapter 5 is

$$\begin{aligned} Q'(\beta,i) \leftarrow\ & \beta{=}\Lambda \wedge i{=}M{+}1 \quad \vee \\ & hd(\beta){=}a_i \wedge Q'(tl(\beta),i{+}1) \quad \vee \\ & hd(\beta) \to \alpha \wedge Q'(\alpha\circ tl(\beta),i) \end{aligned}$$
Compute: $Q'([S],1)$.

By inverting this algorithm, we can obtain a bottom-up (shift-reduce) parser.

Before doing the inversion, it is convenient to modify this algorithm slightly, writing i where there was $i-1$ previously. We can do this by abstracting the body of Q', giving

$$Q'(\beta,i) \leftarrow Q(\beta,i-1),$$

where Q is a new symbol, and unfolding all instances of Q':

$$\begin{aligned} Q(\beta,i) \leftarrow\ & \beta=\Lambda \wedge i=M \quad \vee \\ & hd(\beta)=a_{i+1} \wedge Q(tl(\beta),i+1) \quad \vee \\ & hd(\beta) \rightarrow \alpha \wedge Q(\alpha \circ tl(\beta),i) \\ \underline{\text{Compute}}:\ & Q([S],0) \end{aligned}$$

We now invert the program, obtaining three implications corresponding to the three disjuncts in the body of the backward program.

$$\begin{aligned} & \beta=\Lambda \wedge i=M \quad \supset \quad Q(\beta,i) \\ & hd(\beta)=a_{i+1} \wedge Q(tl(\beta),i+1) \quad \supset \quad Q(\beta,i) \\ & hd(\beta) \rightarrow \alpha \wedge Q(\alpha \circ tl(\beta),i) \quad \supset \quad Q(\beta,i) \\ & \underline{\text{Compute}}:\ Q([S],0) \end{aligned}$$

Straightforward simplification (using δ for $tl(\beta)$) yields

$$\begin{aligned} & \textbf{true} \quad \supset \quad Q(\Lambda,M) \\ & Q(\delta,i) \quad \supset \quad Q([a_i] \circ \delta, i-1) \\ & Q(\alpha \circ \gamma, i) \wedge A \rightarrow \alpha \quad \supset \quad Q([A] \circ \gamma, i) \\ & \underline{\text{Compute}}:\ Q([S],0). \end{aligned}$$

This is a basic bottom-up parser: The first argument to Q corresponds to the stack, and the second to the pointer in the input string. Note that this program terminates for non-loop-free grammars, whereas the initial top-down program does not. (See the discussion in Section 2.)

Further development. The bottom-up parser can be developed a bit further through the use of techniques that have not been described previously in this work. We sketch these steps here.

If we are willing to sacrifice the stronger termination properties, then we could convert the basic bottom-up program directly back into a backward

form, *without* inverting. We would obtain

$$\begin{aligned} Q(\delta,i) \leftarrow\ & \delta=[S] \wedge i=0 \quad \vee \\ & Q([a_i] \circ \delta, i-1) \quad \vee \\ & \delta=\alpha \circ \gamma \wedge A \rightarrow \alpha \wedge Q([A] \circ \gamma, i) \\ \underline{\text{Compute}}:\ & Q(\Lambda,M). \end{aligned}$$

(Justification of this transformation requires close reasoning about the actions of the forward program above.) This program is a bottom-up parser, but because of the inversion step, it parses from right to left.

Had we started with a right-to-left version of the top-down algorithm, we would have obtained the program

$$\begin{aligned} Q(\delta,i) \leftarrow\ & \delta=[S] \wedge i=M+1 \quad \vee \\ & Q([a_i] \circ \delta, i+1) \quad \vee \\ & \delta=\alpha \circ \gamma \wedge A \rightarrow \alpha \wedge Q([A] \circ \gamma, i) \\ \underline{\text{Compute}}:\ & Q(\Lambda,1). \end{aligned}$$

This program very clearly reflects the structure of the shift-reduce parser: As before, δ represents the stack and i the current location in the final string. Initially, the stack is empty and the pointer is at the beginning of the final string. The first disjunct is true if the parse is complete — if the stack contains only the start symbol and the input pointer is at the right end of the final string. The second disjunct corresponds to the *shift* step, in which a symbol is moved from the final string onto the stack and the input pointer incremented. The final disjunct corresponds to the *reduce* step, in which a prefix of the stack that is also the right-hand side of a production is replaced by the left-hand side of that production.

There are two points of nondeterminism in this program: The first choice is whether to shift or reduce; the second is which production to apply when reducing.

Since this algorithm is in linear form, we can convert it into imperative form using the techniques outlined in Section 1. Because the disjunction is

parallel, the resulting program is nondeterministic.

```
given [a_1, ..., a_M]
Q       δ,1 := Λ,1
        if δ = [S] ∘ i = M + 1 then exit
        if choose() then goto reduce
shift:  δ,i := [a_i] ∘ δ, i + 1
        goto Q
reduce: α ∘ γ ⇐ δ
        A ⇐ {A | A → α}
        δ := [A] ∘ γ
        goto Q
```

This program makes use of two backtracking primitives to simulate nondeterminism. The first is the *choose* function of no arguments, which chooses a truth value and returns it. In this example, the choice is whether to shift or reduce. The second is nondeterministic assignment, written " $\Leftarrow$," in which the value assigned is chosen from a finite number of possible values. The first instance of this is the assignment $\alpha \circ \gamma \Leftarrow \delta$, in which a prefix α of the stack δ is chosen for reduction. The meaning of the second instance is clear: Choose the left-hand side of a production whose right-hand side is α.

Failure is obtained if there are no choices at a particular choice point. For example, an α could be chosen such that there is no production $A \rightarrow \alpha$. In the event of failure, the system backtracks to the previous choice point evaluated and makes another choice. If there are no more choices at this point, then another failure is propagated to the next higher choice point, and so on. Our convention is that if all choices fail (i.e., the evaluation ends in failure), then the value of Q is **false**. Otherwise, the **exit** statement will be evaluated, indicating that the result of Q is true and that the parse is successful.

Bibliography

[Aho73] A.V. Aho and J.D. Ullman, **The Theory of Parsing, Translation, and Compiling.** Prentice-Hall, 1973 (two volumes).

[Aho74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman, **The Design and Analysis of Computer Algorithms.** *Addison-Wesley*, 1974.

[Aiello78] L. Aiello, G. Attardi, and G. Prini, "Towards a More Declarative Programming Style." *Formal Descriptions of Programming Concepts*, E.J. Neuhold, ed., North-Holland, 1978.

[Arsac79] J.J. Arsac, "Syntactic Source to Source Transforms and Program Manipulation." *Communications of the ACM*, Vol. 22, No. 1, January 1979.

[Aubin75] R. Aubin, "Some Generalization Heuristics in Proofs by Induction." *IRIA Symposium on Proving and Improving Programs.*, Arc-et-Senans, France, July 1975.

[Balzer72] R.M. Balzer, "Automatic Programming." USC Information Science Institute Report, September 1972.

[Balzer77] R.M. Balzer, "Informality on Program Specifications." *Fifth International Joint Conference on Artificial Intelligence*, Cambridge, August 1977.

[Barstow77] D. Barstow, "A Knowledge Based System for Automatic Program Construction." *Fifth International Joint Conference on Artificial Intelligence*, Cambridge, August 1977.

[Barstow80] D. Barstow, "Remarks on 'A Synthesis of Several Sorting Algorithms' by John Darlington." *Acta Informatica*, Vol. 13, page 225, 1980.

[Bates79] J.L. Bates, "A Logic for Correct Program Development." Ph.D. Thesis, Cornell University, August 1979.

[Bauer78] F.L. Bauer, "Design of a Programming Language for a Program Transformation System." Munich Summer School, August 1978.

[Bird77a] R.S. Bird, "Notes on Recursion Elimination." *Communications of the ACM*, Vol. 20, No. 6, June 1977.

[Bird77b] R.S. Bird, "Improving Programs by the Introduction of Recursion." *Communications of the ACM*, Vol. 20, No. 11, November 1977.

[Boyer75] R.S. Boyer and J.S. Moore, "Proving Theorems about LISP Functions." *Journal of the ACM*, Vol. 22, No. 1, January 1975.

[Burstall71] R.M. Burstall, J.S. Collins, and R.J. Popplestone, **Programming in POP-2.** Edinburgh University Press, 1971.

[Burstall77] R.M. Burstall and J. Darlington, "A Transformation System for Developing Recursive Programs." *Journal of the ACM*, Vol. 24, No. 1, January 1977.

[Burstall78] R.M. Burstall and M.S. Feather, "Program Development by Transformation: An Overview." *Toulouse CREST Course of Programming*, Toulouse, 1978.

[Cadiou72] J.M. Cadiou, "Recursive Definitions of Partial Functions and their Computations." Stanford Computer Science Report, March 1972.

[Cartwright79] R. Cartwright and J. McCarthy, "First Order Programming Logic." *Sixth Symposium on Principles of Programming Languages*, San Antonio, January 1979.

[Chatelin77] P. Chatelin, "Self-redefinition as a Program Manipulation Strategy." *Symposium on Artificial Intelligence and Programming Languages*, Rochester, August 1977.

[Cheatham72] T.E. Cheatham and B. Wegbreit, "A Laboratory for the Study of Automatic Proramming." *AFIPS 1972 Spring Joint Computer Conference*, Vol. 40, 1972.

[Cheatham79] T.E. Cheatham, G.H. Holloway, and J.A. Townley, "Symbolic Evaluation and the Analysis of Programs." *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 4, July 1979.

[Clark77] K. Clark and S. Sickel, "Predicate Logic: A Calculus for Deriving Programs." *Fifth International Joint Conference on Artificial Intelligence*, Cambridge, August 1977.

[Clark80] K. Clark and J. Darlington, "Algorithm Classification through Synthesis." *Computer Journal*, Vol. 23, No. 1, February 1980.

[Cocke71] J. Cocke and J. Schwartz, "Programming Languages and their Compilers." Courant Institute, April 1970.

[Cocke77] J. Cocke and K. Kennedy, "An Algorithm for the Reduction of Operator Strength." *Communications of the ACM*, Vol. 20, No. 11, November 1977.

[Cohen79] N.H. Cohen, "Characterization and Elimination of Redundancy in Recursive Programs." *Sixth Symposium on Principles of Programming Languages*, San Antonio, January 1979.

[Correll78] C.H. Correll, "Proving Programs Correct through Refinement." *Acta Informatica*, Vol. 9, page 121, 1978.

[Dahl72] O.J. Dahl, E.W. Dijkstra, and C.A.R. Hoare, **Structured Programming.** Academic Press, 1972.

[Darlington72] J. Darlington, "A Semantic Approach to Automatic Program Improvement." Ph.D. Thesis, University of Edinburgh, 1972.

[Darlington74] J. Darlington and R.M. Burstall, "A System which Automatically Improves Programs." *Acta Informatica*, Vol. 6, page 41, 1976.

[Darlington78a] J. Darlington, "A Synthesis of Several Sorting Algorithms." *Acta Informatica*, Vol. 11, page 1, 1978.

[Darlington78b] J. Darlington and R. Waldinger, "Case Studies in Program Transformation and Synthesis." SRI International Report, 1978.

[DeBakker76] J.W. DeBakker, "Least Fixed Points Revisited." *Theoretical Computer Science*, Vol. 2, page 155, 1976.

[Demers80] A. Demers and J. Donahue, "'Type Completeness' as a Language Principle." *Seventh Symposium on Principles of Programming Languages*, January 1980.

[Dershowitz79a] N. Dershowitz and Z. Manna, "Proving Termination with Multiset Orderings." *Communications of the ACM*, Vol. 22, No. 8, August 1979.

[Dershowitz79b] N. Dershowitz, "A Note on Simplification Orderings." University of Illinois Report, April 1979.

[Dewar79] R. Dewar, M. Sharir, and E. Weixelbaum, "Transformational Derivation of a Garbage Collection Algorithm." Courant Institute of Mathematical Sciences, New York University, April 1979.

[Dijkstra75] E.W. Dijkstra, "Guarded Commands, Nondeterminacy and Formal Derivation of Programs." *Communications of the ACM*, Vol. 18, No. 8, August 1975.

[Dijkstra76] E.W. Dijkstra, **A Discipline of Programming. Prentice-Hall,** 1976.

[Dijkstra77] E.W. Dijkstra, "Programming: From Craft to Scientific Discipline." *Intl. Comp. Symp.*, F. Morlet and D. Ribbens, eds., 1977.

[Earley68] J. Earley, "An Efficient Context-Free Parsing Algorithm." Ph.D. Thesis, Carnegie-Mellon Univ., Pittsburgh, 1968.

[Earley70] J. Earley, "An Efficient Context-Free Parsing Algorithm." *Communications of the ACM*, Vol. 13, No. 2, February 1970.

[Earley74] J. Earley, "High Level Operations in Automatic Programming." *Symposium on Very High Level Languages, SIGPLAN Notices*, Vol. 9, No. 4, 1974.

[Earnest74] C. Earnest, "Some Topics in Code Optimization." *Journal of the ACM*, Vol. 21, No. 1, January 1974.

[Feather79] M.S. Feather, "A System for Developing Programs by Transformation." Ph.D. Thesis, University of Edinburgh, 1979.

[Gerhart75] S.L. Gerhart, "Correctness Preserving Program Transformations." *Second Symposium on Principles of Programming Languages*, Palo Alto, January 1975.

[Gerhart76] S.L. Gerhart, "Proof Theory of Partial Correctness Verification Systems." *SIAM Journal of Computing*, Vol. 5, No. 3, September 1976.

[Green78] C.C. Green and D.R. Barstow, "On Program Synthesis Knowledge." *Artificial Intelligence*, Vol. 10, page 241, 1978.

[Guttag78] J.V. Guttag, E. Horowitz, D.R. Musser, "Abstract Data Types and Software Validation." *Communications of the ACM*, Vol. 21, No. 12, December 1978.

AD-A091 187 STANFORD UNIV CA DEPT OF COMPUTER SCIENCE F/G 12/1
EXPRESSION PROCEDURES AND PROGRAM DERIVATION.(U)
AUG 80 W L SCHERLIS N00014-76-C-0687
UNCLASSIFIED STAN-CS-80-818 NL

[Haraldsson77] A. Haraldsson, "A Program Manipulation System Based on Partial Evaluation." Ph.D. Thesis, Linköping University, Sweden 1977.

[Hopcroft69] J.E. Hopcroft and J.D. Ullman, **Formal Languages and their Relation to Automata.** Addison-Wesley, 1969.

[Huet78] G. Huet and B. Lang, "Proving and Applying Program Transformations Expressed with Second-Order Patterns." *Acta Informatica*, Vol. 11, page 31, 1978.

[Ichbiah79] J.D. Ichbiah, J.G.P. Barnes, J.C. Heliard, B. Krieg-Brueckner, O. Roubine, and B.A. Wichmann, "Rationale for the Design of the ADA Programming Language." *SIGPLAN Notices*, Vol. 14, No. 6B, June 1979.

[Jensen74] K. Jensen and N. Wirth, **PASCAL User Manual and Report.** Second edition, Springer-Verlag, 1974.

[Katz78] S. Katz, "Program Optimization Using Invariants." *IEEE Transactions on Software Engineering*, Vol. SE-4, No. 5, September, 1978.

[Kleene71] S.C. Kleene, **Introduction to Metamathematics.** North-Holland, 1971.

[Kleene79] S.C. Kleene, "Origins of Recursive Function Theory." *Twentieth Symposium on the Foundations of Computer Science*, 1979.

[Knuth74a] D.E. Knuth and J.L. Szwarcfiter, "A Structured Program to Generate all Topological Sorting Arrangements." *Information Processing Letters*, Vol. 2, page 153, 1974.

[Knuth74b] D.E. Knuth, "Structured Programming with go to Statements." *Computing Surveys*, Vol. 6, No. 4, December 1974.

[Kowalski79] R. Kowalski, "Algorithm = Logic + Control." *Communications of the ACM*, Vol. 22, No. 7, July 1979.

[Kott78] L. Kott, "About R. Burstall and J. Darlington's Transformation System: A Theoretical Study." Université Paris VII, 1978.

[Landin66] P.J. Landin, "The Next 700 Programming Languages." *Communications of the ACM*, Vol. 9, No. 3, March 1966.

[Lansweerde78] A. van Lansweerde, "From Verifying Termination to Guaranteeing It: A Case Study." *Formal Descriptions of Programming Concepts*, E.J. Heuhold, ed., North-Holland, 1978.

[Liskov75] B.H. Liskov and S.N. Zilles, "Specification Techniques for Data Abstractions." *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, March 1975.

[Liskov77] B.H. Liskov, A. Snyder, R. Atkinson, and C. Schaffert, "Abstraction Mechanisms in CLU." *Communications of the ACM*, Vol. 20, No. 8, August 1977.

[Loveman77] D.B. Loveman, "Program Improvement by Source to Source Transformation." *Journal of the ACM*, Vol. 24, No. 1, January 1977.

[Manna73] Z. Manna, S. Ness, and J. Vuillemin, "Inductive Methods for Proving Properties of Programs." *Communications of the ACM*, Vol. 16, No. 8, August 1973.

[Manna74] Z. Manna, **Mathematical Theory of Computation.** McGraw-Hill, 1974.

[Manna75] Z. Manna and R. Waldinger, "Knowledge and Reasoning in Program Synthesis." *Artificial Intelligence*, Vol. 6, page 175, 1975.

[Manna79] Z. Manna and R. Waldinger, "Synthesis: Dreams ⇒ Programs." *IEEE Transactions on Software Engineering*, Vol. SE-5, No. 4, July 1979.

[Manna80] Z. Manna and R. Waldinger, "A Deductive Approach to Program Synthesis." *ACM Transactions of Programming Languages and Systems*, Vol. 2, No. 1, January 1980.

[Manna81] Z. Manna and R. Waldinger, Deductive Aspects of Computer Programming. To appear, 1981.

[McCarthy78] J. McCarthy and C. Talcott, "LISP Programming and Proving." Course notes, Stanford University, 1978.

[McKeeman75] W.M. McKeeman, "On Preventing Programming Languages from Interfering with Programming." *IEEE Transactions on Software Engineering*, Vol. SE-1, No. 1, March 1975.

[Minsky70] M. Minsky, "Form and Content in Computer Science." *Journal of the ACM*, Vol. 17, No. 2, April 1970.

[Paige77] R. Paige and J.T. Schwartz, "Reduction in Strength of High Level Operations." *Fourth Symposium on Principles of Programming Languages*, Los Angeles, January 1977.

[Paige79] R. Paige, "Expression Continuity and the Formal Differentiation of Programs." Courant Computer Science Report 15, September 1979.

[Péter51] R. Péter, Rekursive Funktionen. Akademischer Verlag, Budapest, 1951.

[Peterossi77] A. Peterossi, "Transformation of Programs and Use of 'Tupling Strategy'." *Informatica 77 Conference*, Bled, Yugoslavia, 1977.

[Schonberg79] E. Schonberg, J.T. Schwartz, and M. Sharir, "Automatic Data Structure Selection in SETL." *Sixth Symposium on Principles of Programming Languages*, January 1979.

[Schwartz73] J.T. Schwartz, "On Programming: The SETL language and Examples of its Use." Courant Institute of Mathematical Sciences, New York University, October 1973.

[Sethi74] R. Sethi, "Testing for the Church-Rosser Property." *Journal of the ACM*, Vol. 2, No. 4, October 1974.

[Sickel79] S. Sickel, "Invertibility of Logic Programs." *Fourth Workshop on Automated Deduction*, Austin, February 1979.

[Spitzen78] J.M. Spitzen, K.N. Levitt, L. Robinson, "An Example of Hierarchical Design and Proof." *Communications of the ACM*, Vol. 21, No. 12, December 1978.

[Standish76a] T.A. Standish, D.C. Harriman, D.F. Kibler, and J.M. Neighbors, "The Irvine Program Transformation Catalogue." USC Information Science Institute Report, January 1976.

[Standish76b] T.A. Standish, "An Example of Program Improvement using Source-to-Source Transformations." *Computer Science Conference*, Anaheim, February 1976.

[Steele76] G.L. Steele and G.J. Sussman, "LAMBDA: The Ultimate Imperative." Massachusetts Institute of Technology Report, March 1976.

[Strong71] H.R. Strong, Jr., "Translating Recursion Equations into Flowcharts." *Journal of Computer and System Sciences*, Vol. 5, No. 3, June 1971.

[Tarjan77] R.E. Tarjan, "Complexity of Combinatorial Algorithms." Stanford Computer Science Report, April 1977.

[Vuillemin73] J. Vuillemin, "Proof Techniques for Recursive Programs." Stanford Computer Science Report, October 1973.

[Wegbreit73] B. Wegbreit, "Procedure Closure in EL1." Center for Research in Computing Technology, Harvard University, May 1973.

[Wegbreit76] B. Wegbreit, "Goal-Directed Program Transformation." *Third Symposium on Principles of Programming Languages*, Tucson, January 1976.

[Weyhrauch79] R.W. Weyhrauch, "Prolegomena to a Mechanised Theory of Formal Reasoning." *Artificial Intelligence*, Vol. 13, page 133, 1980.

[Wirth71] N. Wirth, "*Program Development by Stepwise Refinement.*" *Communications of the ACM*, Vol. 14, No. 4, April 1971.

[Wirth76] N. Wirth, "MODULA: A Language for Modular Programming." Technische Hochschule Zürich, March 1976.

[Wulf76] W.A. Wulf, R.L. London, and M. Shaw, "An Introduction to the Construction and Verification of Alphard Programs." *IEEE Transactions on Software Engineering*, Vol. SE-2, No. 4, December 1976.