Debugging Programs After Structure-Changing Transformation

by

Rickard Edward Faith

A Dissertation submitted to the faculty of The University of North Carolina at Chapel Hill in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the Department of Computer Science.

Chapel Hill

1997

Approved by:

Jan F. Prins, Advisor

Siddhartha Chatterjee, Reader

Lars S. Nyland, Reader

RICKARD EDWARD FAITH. Debugging Programs After Structure-Changing Transformation.
(Under the direction of Jan F. Prins.)

ABSTRACT

Translators convert a program from one language to another, and are used to solve a wide range of problems, such as the construction of compilers, optimizers, and preprocessors. Although many tools support the creation of translators, these tools do not provide integrated support for debugging the translator or the output of the translator.

This dissertation investigates the *tracking* of information necessary to provide debugging capabilities for those translators that are structured as a set of program transformations operating on a tree-based representation. In this setting I describe how basic debugging capabilities can be automatically and transparently defined without semantic knowledge of the languages being translated. Furthermore, advanced debugging support, relying on the semantics of the languages and transformations, can be incorporated into this basic framework in a systematic manner.

To evaluate this approach I have constructed KHEPERA, a program transformation system with integral support for the construction of debuggers. With this system I have explored debugging capabilities for traditional compiler optimizations, for more aggressive loop and parallelizing transformations, and for the transformation process itself. I also present algorithms that increase the performance of the transformation process.

In Memoriam

Eva D. Faith
February 9, 1922 – March 17, 1985

Edward S. Faith
August 27, 1913 – March 7, 1987

Acknowledgments

Special thanks to my advisor, Jan F. Prins; my readers, Siddhartha Chatterjee and Lars S. Nyland; and the other members of my committee, Peter Calingaert, John McHugh, and Don Stanat.

During my stay in the Computer Science Department at the University of North Carolina, I have been supported by Susanna Schwab, David V. Beard, and Jan F. Prins. Much of the work described in this dissertation was funded by ARPA via ONR contract N00014-92-C-0182, by Rome Labs contract F30602-94-C-0037, and by a Cray Fellowship from Cray Research, Inc. and NCSC (North Carolina Supercomputing Center).

Thanks to the friends and colleagues who I have worked with while at UNC: Doug L. Hoffman, Kevin E. Martin, Jonathan P. Munson, Daniel L. Palmer, M. Paramasivam, James W. Riely, Stephen G. Tell, and Bill Yakowenko.

My most profound and heartfelt thanks is reserved for my wife, Melissa, and daughter, Rhiannon.

Contents

List of Tables

List of Figures

List of Abbreviations

AST	abstract syntax tree
CFG	context free grammar
CSE	common-subexpression elimination
DAG	directed, acyclic graph
DSL	domain-specific langauge
IR	intermediate representation

Glossary

Throughout this dissertation, several words are used to convey specific meanings or to draw specific distinctions. This glossary is provided as a convenient reference for the reader.

Abstract Syntax Tree AST is used to refer to any tree-based representation of a program, even if that representation does not strictly embody abstract syntax.

Compiler A common, off-the-shelf program that converts a program written in a high-level language into exectuable code. A compiler is a special case of a program translator.

Debugger A program used to debug other programs. "Debugger" never refers to a human.

Optimization A transformation that tends to improve the time or space requirements for a program.

Source Language The input language for a translator.

Target Language The output language for a translator.

Transformation A relation between two programs that is valid if and only if the programs are semantically equivalent.

Unoptimized Program A program that was compiled in a straight-forward manner, such that there is a simple relationship between the machine code generated and the statements in the program from which it is generated.

Chapter 1

Problem Definition

1.1 The Problem

Translators are used to convert a program from one language to another. A compiler, which translates a program from a high-level language into assembly code, is the best known type of translator. Translators are also used to solve a wide range of problems beyond compilation, including the support for programming language extensions and preprocessors. Although many tools are available that support the creation of translators, these tools provide little support for debugging the translator or the output of the translator.

This dissertation investigates the *tracking* of information necessary to provide debugging capabilities for those translators that are structured as a set of program transformations operating on a tree-based representation. In this setting I describe how basic debugging capabilities can be automatically and transparently defined without semantic knowledge of the languages being translated. Furthermore, advanced debugging support that relies on the semantics of the languages and transformations, can be incorporated into this basic framework in a systematic manner.

1.2 Motivation

This section outlines the widespread use of translators and the problems providing debugging support for translator output and for the translator itself.

$$\text{Program written in } L \longrightarrow \boxed{\text{Translator}} \xrightarrow{\text{C}} \boxed{\text{C Compiler}} \longrightarrow \begin{array}{l}\text{Native}\\ \text{Executable}\end{array}$$

Figure 1.1: Structure of Multistage Translator

1.2.1 Pervasive Use of Program Translators

Researchers often implement compilers for a new language L as translators from L to an existing language. Compilers performing optimizations or parallelization are also frequently implemented as source-to-source translators. In these cases, the overall process of compilation consists of the composition of translators, as shown in Figure 1.1. The ability to compose translators into a single "multistage" translation system provides several advantages, among them:

- Ease of implementation: the native high-level language compiler takes care of low-level, machine-specific details.

- Portability: a high-level language, such as C or FORTRAN 90, can be viewed as a "portable assembly language".

- Efficiency: the native compiler provides machine-specific optimizations and an interface to the operating system, freeing the researcher to concentrate on the research language or optimization techniques being explored.

For example, implementations of the SISAL [Cann 1992], pC++ [Gannon et al. 1994], PROTEUS [Prins and Palmer 1993], and Mercury [Henderson et al. 1995] languages all use the native C or C++ compiler on the target machine as a back end for the compilation process. At least one implementation of a High Performance FORTRAN (HPF) compiler generates FORTRAN 77 output [Bozkus et al. 1995]. Other systems, like the Parafrase-2 [Polychronopoulos et al. 1990] parallelizing compiler, are also implemented as a program translator composed with a compiler for a high-level language.

Program translators are also used outside the research environment for implementing new languages (e.g., AT&T `cfront` [Stroustrup 1994], MODULA-3 [Harbison 1990], EIFFEL [Meyer 1988]), for maintaining backward compatibility with old languages (e.g., the FORTRAN-to-C translator, `f2c` [Feldman et al. 1995]), and for implementing database programming systems that "compile" to C code with library calls [Elmasri and Navathe 1989].

1.2.2 Difficulties Providing Sophisticated Debugging Support

Although program translators have been used to implement a wide variety of research and commercial "compilers", including a wide range of source-to-source optimizers, multi-stage compilers, and domain-specific langauge (DSL) processors, the implementations of program translators often lack debugging support. When debugging support is provided, it is often primitive (e.g., in the past, `cfront` output was often debugged with a debugger that did not understand the C++ name-mangling conventions); requires that many interesting optimizations be disabled (e.g., SISAL); or requires recompilation as part of the debugging process (e.g., EIFFEL).

Traditional UNIX[1] implementations for C provide debugging support in the context of a multistage translator. In these implementations, C is compiled into assembly code, optimized by a standalone peep-hole optimizer, and then assembled into object code. Information is transferred between successive stages of the compiler so that symbolic debugging can be supported. This debugging support is usually implemented in an ad hoc fashion for each specific compiler and requires considerable implementation overhead. Even then, the debugger tool usually ignores the effects of optimizations, causing considerable confusion for the programmer who attempts to debug optimized code.

The key problem with providing debugging support for new languages has been the difficulty of implementing the necessary language-specific debugging support. At minimum, this debugging support would provide mappings between the source program and the target language, with a simple interface to the existing debugger for the target language. But even this level of support involves tremendous programmer overhead, especially when the syntax and semantics of a language have not been frozen. Significant work is involved in the implementation of a compiler or translator for any new language—providing support for advanced symbolic debugging may prohibitively increase the complexity of this work, especially for new or experimental languages in a research setting.

[1] UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

1.2.3 Translators Require Specialized Debugging Support

There is a fundamental difference between debugging a translated program (as described in Section 1.2.2, above) and debugging the translation process.

Debuggers generally provide access to the basic data types supported by a language, and allow setting breakpoints at execution points which are reasonable for that language. If a translator is implemented with the C programming language, then the debugger would allow access to data types such as integers and arrays of characters, and allow breakpoints to be set on C statements. The translator, however, is operating at a level of abstraction that is much higher than the C code which implements it. The basic data types in a translator represent abstract objects, such as the intermediate representation (IR) of the program and symbol table entries. Setting a "breakpoint" in a translator should, at a high level of abstraction, interrupt the translation process at some reasonable point within that process (instead of some arbitrary point that the C debugger would produce).

Hence, debugging a translator written in C is fundamentally different from debugging an ordinary C program, and requires specialized debugging support. The debugging algorithms discussed in this dissertation can be used to answer questions about the program being translated or about the translation itself—these questions will be outlined in the next section.

1.3 Approach

In this dissertation, optimizations, parallelization techniques, and language translation will all be viewed as the composition of successive program transformations applied to an abstract syntax tree (AST) [Loveman 1977], as shown in Figure 1.2. The AST, which provides a very general program representation, is commonly used as an IR by compiler and translator implementors [Appel 1997; Muchnick 1997].

Consider the problem of translating a program P, written in the source language L, into a program in the output language L'. In Figure 1.2, T_0 is an AST which represents P after the parsing phase, ρ. T_ℓ is the final transformed AST, and P' is a valid program, in the output language L', constructed from T_ℓ during the unparsing phase, σ. The transformation process is viewed as the composition of successive transformations functions, $k = 1, \ldots, \ell, \tau_{k+1}(T_k) = T_{k+1}$, to the AST. The determination of which transformation function to apply next may require extensive analysis of the AST. Once the transformation functions are determined, however, they can be rapidly

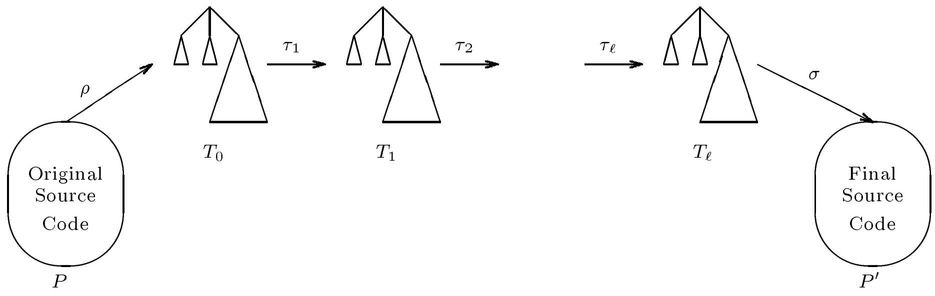


Figure 1.2: Transformation Process

applied for replay or debugging.

When debugging a set of program transformations, the programmer requires detailed information about each successive application of a transformation. Since merely providing snapshots of each intermediate AST would be overwhelming and confusing, methods to examine the AST in different ways are necessary:

- The programmer may need to look at two successive ASTs and view only the updated portions.
- The programmer may need to identify some "interesting" subset of the AST and view only the transformations that involve this part, skipping all other transformations.
- The programmer may need to examine the transformed ASTs in either the forward (e.g., $T_k, T_{k+1}, \ldots$) or in the reverse (e.g., $T_k, T_{k-1}, \ldots$) transformation direction.

This level of debugger functionality is necessary to provide detailed, manageable debugging information for the transformations themselves. Note that debuggers can be composed in the same way that compilers were composed in Figure 1.1, and that if the functions needed for debugging the transformations can be provided, then sufficient information will also be available to provide traditional debugging functions:

- setting breakpoints
- determining current execution location (e.g., in response to a breakpoint or program exception)

- reporting a procedure traceback
- displaying values of variables

The end-user, who is using the language processor to transform programs, may not be interested in viewing the detailed transformation machinery. For this type of end-user, the debugging queries will relate P and P', and will avoid the intermediate tree representations. Based on anecdotal evidence obtained while observing and interacting with scientists working on optimizing the performance of legacy FORTRAN codes, I believe sophisticated end-users of the transformation system will be interested in viewing intermediate tree representations. The sophisticated end-user will use the capabilities of setting breakpoints by selecting semantic elements from an intermediate tree and will want to determine execution location or variable values on an intermediate tree.

1.4 Scope and Goals of this Work

Given a translator implemented as a series of tree transformations:

- I can define *tracking* of information that is *automatic* and *transparent* to the implementor of the translator.
- This tracking enables building simple debuggers *without semantic knowledge* of the languages being transformed.
- This tracking provides a *framework* for building sophisticated debuggers that require semantic knowledge of the transformations.

The work presented here is limited in two ways. First, since a large variety of language processors can be implemented using transformation on a tree-based intermediate representation, my work concentrates on providing debugging capabilities for these translators. Second, since these translators are often implemented as high-level language translators, my work concentrates on high-level language translation as described in Section 1.2.1, and leaves generation of machine code as future work.

In summary, I have concentrated on a framework for debugging in the face of aggressive program transformations. The solution does not require any debugging methods that could perturb the run-time characteristics of the program being debugged. Further, the solution requires minimal assistance from the implementor of the translator, and does not restrict the type or complexity of transformations performed. Compared with other debugging methods, the contributions made in this dissertation include a description of:

1. debugging support that is independent of the semantics of the language being transformed;

2. scaffolding for the support of other debugging methods that do depend on semantic information;

3. debugging support that is transparent to the translator implementor;

4. debugging support for a large class of translators, including those that utilize common scalar transformations, aggressive loop transformations, and complex parallelization transformations; and

5. debugging support for both the output of the translator and for the translator itself.

In addition, I have implemented the proposed debugging algorithms in the KHEPERA transformational programming system; I have written a viewer for KHEPERA that can be used to debug translator implementations; and I have explored the problem of rapid tree-traversal on the AST intermediate representation.

Chapter 2 reviews the previous work on symbolic debugging of optimized programs and explores enhancements necessary for debugging programs that have been aggressively transformed using arbitrary structure-changing transformations on a tree-based intermediate representation.

Chapter 3 outlines specific algorithms for tracking and replaying program transformations.

Chapter 4 demonstrates the practicality of these algorithms by discussing their implementation in KHEPERA, a system for writing and debugging complex transformation systems.

Chapter 5 demonstrates the generality and usefulness of these algorithms by presenting an example of debugging the transformation system itself and by discussing the construction of more sophisticated debuggers using the framework presented here.

Chapter 6 summarizes the work presented in this dissertation, the contributions made, and possible areas for future exploration.

Chapter 2

Related Work

This chapter presents an overview of debugger functionality and the terminology necessary to discuss source-level (or symbolic) debuggers. This discussion will cover the two main categories of debuggers (*expected behavior* and *truthful behavior*) and the associated problems posed by each category in the face of various kinds of optimizations. Common optimizations will not be discussed in detail in this dissertation—curious readers should consult one of the catalogs of optimizations that have been published: Loveman [1977] and Wolfe [1989, 1996] (aggressive loop and vectorization optimizations), Bacon et al. [1994] (an excellent overview of both scalar and vectorization optimizations), and Muchnick [1997] (common scalar optimizations). I will review the previous work in the field of source-level debugging in terms of the problems solved and the optimizations handled. Finally, I will discuss the previous work that is most applicable to the debugging of transformation systems.

2.1 Debugger Functionality

In general, a debugger must be able to perform the following basic tasks:

1. Associate source code positions with machine code locations:

   Set a breakpoint. The user specifies a point in the program at which execution should stop, and the debugger runs the program until this point is reached.

   Trap or breakpoint location reporting. The program stops because of an exception or a user-defined breakpoint, and the debugger reports the position in the original source code at which execution has stopped.

2. Associate source code variables with memory locations:

   Display the value of a variable. The user selects a variable at some position in the program (e.g., at the current breakpoint), and the debugger reports the value of the variable.

   Change the value of a variable. The user selects a variable at some position in the program (e.g., at the current breakpoint), and specifies a new value for this variable. The debugger updates the value.

In addition to the control breakpoints discussed above (which break at a specific point in the control flow of the program), more complicated debuggers may also provide breakpoints that stop execution when a variable is updated (data breakpoints) or when a user-defined predicate becomes true (conditional breakpoints). These advanced breakpoints and other possible debugger features (e.g., the ability to modify the executing code, or to execute arbitrary code during the debugging session) are beyond the scope of this dissertation. These advanced features depend, at the lowest level of debugger functionality, on the ability of the debugger to map source code positions and source code variables to machine code locations and memory locations. Ultimately, these advanced features can be implemented using the basic functionality described above.

Traditionally, the granularity for breakpoints has been at the line or statement level. When dealing with functional languages, or with aggressive optimizations, it may be more convenient to set a breakpoint in the middle of a statement or an expression. The techniques and algorithms described later in this dissertation always consider the source program as a collection of abstract syntactic or semantic elements. From this viewpoint, breakpoints can be set on statements (if the language supports the semantic notion of "statement") or the breakpoint can be set on any other semantic element, such as an assignment or an expression. Traditional debuggers generally report source code locations in terms of the statement boundary because this is the granularity of information provided by the compilation system. If more precise information can be provided, then debuggers should provide that information to the user.

In this discussion, I have used the terms "machine code locations" and "memory locations" to convey the idea that the interesting associations are between high-level source code and the low-level code that is being executed. Later in this dissertation, I will generalize this notion to include associations between high-level "source" code and high-level "target" code (which may or may not be directly executable, but which is the result of transformation or "compilation" of the original source code). For the remainder of this chapter, however, thinking in terms of "source code" which is compiled into "machine code" is a useful convenience.

2.2 Properties of Source-Level Debuggers

Zellweger [1984] defines two general classes of debuggers:

Expected Behavior A debugger that provides expected behavior "...always responds exactly as it would for an unoptimized version of the same program" [Zellweger 1984, p. 34]. The effects of optimizations and program transformation are hidden from the user.

Truthful Behavior A debugger that provides truthful behavior "...avoids misleading the user: it either displays (in source program terms) how optimizations have changed the program portion under consideration, or it admits that it cannot give a correct response" [Zellweger 1984, p. 34]. A truthful debugger may provide a response, but warn that the response may be incorrect.

Copperman and McDowell [1993] investigated several contemporary compiler/debugger combinations and found that none of the examined combinations could provide expected or truthful behavior for all of the example cases. Some combinations did provide expected or truthful behavior for some cases—however, other combinations did not provide expected or truthful behavior for any example cases.

I believe that it is better to depict how a program is actually behaving than to try to hide the effects of optimization. Hiding the effects of optimization is undesirable because the user is prevented from debugging the actual code that is being executed, and must blindly trust that the compiler and debugger have correct implementations for both the optimizations and the machinery which undoes the effects of the optimizations. Further, as the optimizations become more and more complicated, hiding their effects may become intractable.

2.3 Problems of Expected Behavior Debugging

Most of the prior work on symbolic debugging has concentrated on providing expected behavior in the face of the scalar optimizations similar to the ones shown in

Constant folding
Copy propagation
Constant propagation
Common-subexpression elimination
Dead assignment elimination
Dead code elimination
Procedure inlining
Cross-jumping
Strength reduction
Induction variable elimination
Loop-invariant code motion
Code hoisting
Loop unswitching
Loop unrolling
Loop peeling

Figure 2.1: Common Scalar Optimizations

Figure 2.1 (see Appendix B for a brief summary of these optimizations). Based on the required debugger functionality, the problems caused by optimizations fall into two main categories: *location problems* (or *code location problems*) and *data problems* (or *data-value problems*) [Zellweger 1984; Adl-Tabatabai 1996].

Zellweger [1984] provides an excellent overview of these major debugging problems:

Location problems. Contemporary debuggers use a simple one-to-one mapping from program source to object code. Often, optimizations such as dead store elimination or unreachable code elimination remove object code from the resulting executable. Other optimizations (e.g., procedure discovery, cross-jumping, loop-unrolling, and inline procedure expansion) can cause object code to be merged or duplicated, complicating the source to object mapping.

Data-value problems. These problems arise when variable values reported by a contemporary debugger are incorrect because variable assignments are moved or deleted by the optimizations applied. Typical optimizations which cause these problems are: constant propagation, copy propagation, induction variable elimination, and code hoisting.

In the next sections, these problems will be discussed in greater detail.

2.3.1 Location Problems

Source-level debuggers usually use a compiler-generated *line table* for setting breakpoints, resolving exception or breakpoint location, and providing information about the execution context and stack. The line table generally contains tuples associating a single line of source code with a single machine code location [Copperman 1993a].

In the face of the common scalar optimizations shown in Figure 2.1, a simple line table is not sufficient, and requires considerable augmentation to properly describe the mappings. For example, dead code elimination will remove tuples from the line table so that a source line will no longer occur in the source code to machine code mapping. In contrast, *control flow* optimizations, which either merge identical code (e.g., procedure discovery, cross-jumping) or duplicate code (e.g., loop unrolling, procedure inlining), create one-to-many or many-to-one source-code to machine-code associations. Other optimizations, such as instruction scheduling, can cause the effects of two or more source statements to be interleaved in the final machine code.

Optimizations used by vectorizing and parallelizing compilers are shown in Figure 2.2 (see Appendix B for a brief summary of these optimizations). Wolfe [1989] discusses these and other optimizations that are useful when vectorizing loops in scalar programs, when discovering parallelism in sequential programs, and when compiling programs written in a language that supports explicit parallelism. Several of these optimizations are also useful on scalar machines that do not provide much concurrency. For example, loop interchange may help to provide increased locality of array references and decreased paging on a scalar virtual memory machine [Wolfe 1989, p. 105].

Location problems in the face of parallelizing optimizations are similar to, but more complicated than, the location problems present with common scalar optimizations. These optimizations are more likely to require one-to-many, many-to-one, and many-to-many source-code to machine-code associations. Further, when concurrency is supported, or when the source language supports explicit process parallelism, associations may span processors.

2.3.2 Examples of Location Problems

In this section, examples of location problems from the expected-behavior debugging literature will be described. These examples are designed to expose the main problems of debugging in the face of optimizations while being straightforward and

Loop interchange
Loop skewing
Loop reversal
Loop coalescing
Strip mining
Loop tiling
Loop splitting
Loop jamming
Software pipelining

Figure 2.2: Aggressive Loop Optimizations

understandable. These examples will be explored in more detail in Section 3.2.4 (page 39).

Control flow optimizations change the mapping between syntactic elements in the input source code and equivalent or related syntactic elements in the output source code. When the mapping is one-to-one, answering queries from the debugger is relatively straightforward: a simple line table is sufficient to answer the queries. However, in the face of control flow optimizations, the mapping may be many-to-one, one-to-many, or many-to-many. Zellweger [1984] concentrates on two control flow optimizations which have these characteristics:

- Inline procedure expansion: the replacement of a function call with the body of the function being called. This creates a one-to-many mapping between lines in the original function body and all of the points in the program at which the function was inlined. For example, in Figure 2.3 (using C-like syntax), calls to function f are inlined in function g. Given line-level granularity, setting a breakpoint on line 11 (shown in the figure with a box) in the original source code should set a breakpoint on lines 11, 21, and 22 in the transformed source code (shown in the figure with boxes). Ideally, however, granularity would allow the breakpoint to be set *just prior* to the assignment on lines 21 and 22, thereby preserving the notion that the breakpoint was requested *within* function f.

- Cross-jumping: merging identical sections of code into a single section. This creates a many-to-one mapping between the identical sections in the original code and the single section in the cross-jumped code. As the example in Fig-

```
10 int f(int x) {
11     return x + x;
12 }

20 int g(int y, int z) {
21     int a = f(y);
22     int b = f(z);
23     return a + b;
24 }
```

⟶

```
10 int f(int x) {
11     return x + x;
12 }

20 int g(int y) {
21     int a = y + y;
22     int b = z + z;
23     return a + b;
24 }
```

Figure 2.3: Procedure Inlining Transformation

```
30 if (a == b) {
31     x = 1;
32     y = 2;
33     z = 3;
34 } else {
35     x = 2;
36     y = 2;
37     z = 3;
38 }
```

⟶

```
30 if (a == b) {
31     x = 1;
34 } else {
35     x = 2;
38 }
39 y = 2;
40 z = 3;
```

Figure 2.4: Cross-Jumping Transformation

> ure 2.4 shows, setting breakpoint on line 32 *or* line 36 in the original source code should set a breakpoint on line 39 in the transformed source code.
>
> Zellweger [1984] uses silent breakpoints to determine the most recently executed path through the **if** statement, and has the debugger ignore the breakpoint on line 39 unless the path in the selected branch of the code was followed. This refinement is an example of a debugging algorithm which is transformation-specific: it works only with the cross-jumping transformation—a different refinement would have to be devised for every other transformation that leads to a many-to-one mapping between original source code and transformed source code.

When both of these transformations are combined, the results can require a many-to-many mapping. As shown in Figure 2.5, a breakpoint set in the original code on *either* line 43 *or* line 46 will require breakpoints to be set in the transformed code on lines 50, 66, *and* 72.

2.3.3 Data-Value Problems

For displaying or changing the value of a variable, source-level debuggers generally use a symbol table containing tuples associating a symbol's name, type, and size with one or more locations in memory [Copperman 1993a].

Because of optimizations, a variable may reside in different locations during its lifetime (e.g., in memory vs. in a register), making the use of a simple symbol table insufficient to determine the value of the variable. Further, after a variable is dead, a debugger may not be able to determine the value because optimizations removed the final store (e.g., dead store elimination).

Even if the debugger can determine the correct location of a variable, the reported value may not be the value *expected* from inspection of the original source code. Much of the previous work on expected behavior debugging has concentrated on *currency determination* in the face of the optimizations listed in Figure 2.1. Hennessy [1982] introduced the following terms to describe the status of a variable at a particular point in a program:

current The value of the variable is guaranteed to be the same as the expected value in unoptimized code, regardless of the path taken to this point in the program.

noncurrent The value of the variable is guaranteed to be computed from different expressions on all paths. Note that, depending on program inputs and the calculations being performed, the values *might* the the same, but this is merely coincidence.

endangered The value of the variable is computed from the same expressions on some paths, but from different expressions on other paths, so currency cannot be statically determined.

Adl-Tabatabai [1996] uses the term *suspect* to indicate a variable whose currency cannot be determined, and uses *endangered* to indicate a variable is either *noncurrent* or *suspect*. This definition is consistent with Hennessy's, if one considers an endangered variable to be one whose value *may not* correspond to the variable's expected value. This classification system is reasonable and useful when providing expected

```
38 int b;

39 int f(int x) {
40     int a;
41     if (x == 3) {
42         a = 1;
43         b = 2;
44     } else {
45         a = 2;
46         b = 2;
47     }
51     return a;
52 }

60 int g(int c) {
80     int n = f(c);
81     int m = f(d);
82     return n + m;
83 }
```

$\longrightarrow$

```
38 int b;

39 int f(int x) {
40     int a;
41     if (x == 3) {
42         a = 1;
44     } else {
45         a = 2;
47     }
50     b = 2;
51     return a;
52 }

60 int g(int c) {
61     if (c == 3) {
62         n = 1;
63     } else {
64         n = 2;
65     }
66     b = 2;
67     if (d == 3) {
68         m = 1;
69     } else {
70         m = 2;
71     }
72     b = 2;
82     return n + m;
83 }
```

Figure 2.5: Procedure Inlining and Cross-Jumping Transformations

A, B, C: **array** (1..n) of **integer**
T: **integer**
do $i = 1, n$
    $T = A(i) + B(i)$
    $C(i) = C(i) + y * T$
end do

$\longrightarrow$

A, B, C, T': **array** (1..n) of **integer**
$T' = A + B$
$C = C + y * T'$

Figure 2.6: Example Loop Vectorization

behavior debugging, or when providing truthful behavior debugging with expected behavior augmentation.

Code motion, storage overlaying, and copy propagation are examples of optimizations which often cause data-value problems because the location of the variable can be determined, but the value is not the expected value. Optimizations that remove variable references, such as constant propagation and induction variable elimination, may cause a variable to be unknown to the debugger. Variable elimination may be a serious problem for expected behavior debuggers—changing the value of these variables during a debugging session may be useless, and reconstructing their values at may be difficult.

When loop vectorizing optimizations are considered, data-value problems are complicated because the type of the variable may change during the optimization. For example, consider the vectorization, using FORTRAN-like syntax, shown in Figure 2.6. In this example, the temporary scalar variable T is optimized away: only the vector variable T' is available during debugging. Further, under Zellweger's definition of *expected behavior* debugging, the debugger should be able to *single step* through this loop, displaying intermediate values for T.

2.3.4 Examples of Data-Value Problems

In this section, examples of data-value problems from the expected behavior debugging literature will be described. These examples are designed to expose the main problems of debugging in the face of optimizations while being straightforward and understandable.

In general, data-value problems are caused by assignments which are either deleted (e.g., via redundant or dead assignment elimination) or moved (e.g., via code hoisting) by the transformations. Examples of these sorts of problems will be presented below,

```
100 ...                    →  100 ...
110 x = y + z;                110 x = y + z;
120 ...                       120 ...
130 x = y + z;
140 ...                       140 ...
```

Figure 2.7: Redundant Assignment Elimination

```
200 ...                    →  200 ...
210 x = w - v;
220 ...                       220 ...
230 x = y + z;                230 x = y + z;
240 ...                       240 ...
```

Figure 2.8: Dead Assignment Elimination

together with an outline of expected behavior solutions. In Section 5.2.3 (page 123), these examples will be revisited, with a discussion of how to use the methods described in this dissertation to debug them.

2.3.4.1 Redundant Assignment Elimination

An example of redundant assignment elimination is shown in Figure 2.7. Here, line 130 is removed because x, y, and z were not modified since the assignment in line 110.

Adl-Tabatabai [1996] points out that this case does not create endangered variables, and so does not impact expected behavior debugging: the value of x did not change, so the elimination of the second assignment does not produce an unexpected value when x is queried at a breakpoint on line 140.

2.3.4.2 Dead Assignment Elimination

An example of dead assignment elimination is shown in Figure 2.8, where x is not used between line 210 and 230.

Given a breakpoint on line 220, the value of x is noncurrent in the final transformed program (since the assignment to x on line 210 was removed). Adl-Tabatabai

```
300 x = u - v;
310 if (c) {
320    x = y + z;
330 } else {
340    ...

360    ...
370 }
380 ...
390 x = y + z;
400 ...
```

$\longrightarrow$

```
300 x = u - v
310 if (c) {
320    x = y + z
330 } else {
340    ...
350    x = y + z;
360    ...
370 }
380 ...

400 ...
```

Figure 2.9: Code Hoisting

[1996] introduces a *dead assignment descriptor* whenever a local dead code elimination eliminates an assignment because of a later assignment. This descriptor helps to detect noncurrency, but at the cost of additional overhead for the implementor of the translator, since the semantics of *assignment* must be understood by the translator when the descriptor is created.

2.3.4.3 Code Hoisting

An example of code hoisting is shown in Figure 2.9, where x is not used during the **else** part.

There are two interesting breakpoints in this example:

Line 360 The expected value of x is $u - v$, but the actual value is $y + z$.

Line 380 The expected value of x is *either* $u - v$ or $y + z$, depending on which branch of the **if** was taken. The actual value is $y + z$. This may lead the programmer to believe that the first branch was always taken, when, in fact, the transformations make this assumption incorrect.

Using algorithms from Adl-Tabatabai [1996], an expected value debugger would report the value of x at breakpoint on line 360 as noncurrent, and the value of x at a breakpoint on line 380 as suspect. Using the methods from Zellweger [1984], a silent breakpoint would be inserted in each branch of the **if** statement, thereby helping the debugger to make a currency determination based on information collected at run-time via the use of program instrumentation.

Apply-to-all elimination
Promotion of functions

Figure 2.10: Transformations for Flattening Nested-Data Parallelism

2.3.5 Difficulties of Providing Expected Behavior

As discussed above, expected behavior debugging has been shown to be possible in the face of most of the optimizations shown in Figure 2.1. In general, these optimizations move or eliminate assignments, and the problem of expected behavior debugging is to undo the effects of the code motion or to otherwise reconstruct the assigned values. Indeed, Adl-Tabatabai [1996] notes:

> ...there are a number of invariants that are preserved when compilers transform programs—compilers do not perform arbitrary transformations; my algorithms take advantage of the invariants maintained by transformations that move or eliminate assignments. For example, if an assignment is hoisted to a different basic block, this basic block is post-dominated by the original block; this limits the range of breaks where a variable is endangered because of the hoisted assignment. Or, if an assignment is eliminated because of backward redundancy, the value must be available somewhere, and the debugger can provide this value to the user.

For other classes of optimizations, such as those shown in Figure 2.2 and Figure 2.10 (see Appendix B for a brief summary of these optimizations), expected behavior debugging poses more complicated problems. Vectorization may spread the computation of a variable over several lines, intermingle several computations that were implicitly serialized in the original source code. The flattening of nested-data parallelism [Blelloch 1990] can be viewed as a transformation that *changes the type* of variables, promoting scalar variables to vectors, or promoting nested sequences to more deeply nested sequences. Reconstructing loops or source-level sequences for expected behavior debugging involves considerable complicated work for the debugger.

As of 1997, only limited, speculative work has been done on solving the problems of providing expected behavior debugging for vectorizing, parallelizing, and flattening transformations. Further, it is not clear that expected behavior would be helpful in the face of these aggressive optimizations, especially for research compilers which are actively being developed.

In the best situation, the optimizations are correctly implemented and the debugger's "undoing" is correct. In this case, an expected behavior debugger might hide critical information about the optimizations from the end-user. Because the optimizations are aggressive and unfamiliar, it may be important for the user to understand how the code was transformed. This is obviously important if the user is also the implementer of the research compiler. Less obviously, this is important for users of high-performance computers who are interested in obtaining the best possible optimizations. These users might be interested in trying more aggressive optimizations, and may feel more confident doing so if they have some understanding of how their code is being transformed.

There are two other cases in which expected behavior debugging is potentially harmful:

1. the optimizations are correctly implemented, but the debugger's "undoing" is incorrect, or

2. the optimizations are incorrectly implemented.

In these cases, the debugger might hide implementation errors and prolong the debugging process instead of making it shorter. From the standpoint of the compiler writer, the work of implementing novel, aggressive transformations is doubled if expected behavior debugging must also be supported: the transformations must be implemented correctly, and the ability of the debugger to undo or hide the effects of the transformations must also be implemented correctly. Errors in either part of this task can hide errors in the other part.

2.4 Problems of Truthful Behavior Debugging

In situations where expected behavior is difficult or impossible to provide, truthful behavior is sometimes considered "the next best thing". In the face of novel, aggressive loop or flattening transformations, the reasons outlined in the previous section indicate truthful behavior is actually the best thing that can be provided.

The main problem with providing useful truthful behavior is that there may be a tremendous amount of information that the debugger must present to the user in order to explain the optimizations. The CXdb debugger [Brooks et al. 1992; Streepy, Jr. 1994] highlights regions of the original source code to show the progress of program execution and to explain the effects of optimizations. Since only the *original* source code is displayed, the mappings between the executing code and the original sources may present overall changes that are too complicated for the user to understand, especially in the face of many composed transformations. Cool [1992] suggests displaying a high-level representation of the original source code and the final transformed output of the translator, using highlighting to indicate which portions of the program have been executed. Cool limits his proposal (there does not appear to be an implementation) to optimizations for software pipelining and loop unrolling.

The debugging system described in this dissertation solves the problem of explaining the transformations to the user by allowing the user to view program snapshots throughout the transformation process. These views can be presented at several different levels of abstraction, thereby allowing the user to ignore the minutiae of the transformations while still obtaining a view of the transformation *process* at several important transformational points. Note that, for the naïve end-user, a view of the original and final transformed versions may be all that is desirable. For a sophisticated end-user, or for the transformation implementor, a view of intermediate transformations may be required. This dissertation shows how to provide these capabilities without forcing a single debugging paradigm on the end-user or the implementor of the program translator.

2.5 Expected Behavior Debugging of Optimized Programs

2.5.1 Manual Recompilation

The simplest way to provide "expected behavior" debugging of optimized code is to require that the programmer manually recompile the source code *with optimizations disabled*, and then execute and debug the unoptimized program. Unfortunately, this is the only way to obtain anything approaching expected behavior debugging using most contemporary production-quality debugging systems. I mention it here for completeness.

Copperman [1993a] discusses two main reasons why disabling optimization is not an acceptable debugging alternative: some languages have semantics which allow multiple correct translations for a certain construct (e.g., the simplification of floatingpoint expressions) so the behavior of a correct program may be different with and without optimization; and a program with a bug may have different behaviors with and without (correct) optimizations. For our work on PROTEUS [Prins and Palmer 1993], and for other work on research compilers, it may be impossible to disable optimizations and still produce an executable program. In the case of PROTEUS, the "optimizations" are actually source-to-source transformations which are an intimate part of the compilation process. In the case of other languages, a similar situation may occur (perhaps the code must be serialized or parallelized before it can execute on the target architecture).

Other languages (e.g., EIFFEL [Meyer 1988]) require that the program be recompiled with a debugging module before debugging can take place. This poses many of the same problems as recompilation without optimization, and adds the additional possibility of the included debugging subsystem perturbing the program, eliminating or changing the behavior that is being debugged.

2.5.2 Restricted Optimizations

Another common approach to the debugging problem is to restrict the set of optimizations that are allowed. This is also a common technique for increasing debuggability in both production compilers (e.g., `gcc`) and research compilers. For example, when debugging is performed, the SISAL compiler [Cann 1992] requires disabling interesting and desirable parallelizing optimizations.

As will be discussed below, even some of the more aggressive techniques for providing expected behavior debugging require limiting the optimizations performed by the compiler. As research has progressed, methods for avoiding limitations have been described. However, aggressive loop, vectorizing, parallelizing, and flattening optimizations still have to be restricted.

2.5.3 On-the-Fly Deoptimization

Pollock and Soffa [1988] propose a program representation that allows the debugger to derive the unoptimized program when necessary. Optimizations and notations for code that has been eliminated, moved, or replaced are made in an *optimization history*. Debugging queries are specified by the user, and portions of the program are then recompiled with some optimizations disabled. Debugging is performed on the new executable which contains some mix of fully optimized, partially optimized, and unoptimized code.

Hölzle et al. [1992] discuss a technique for *dynamic deoptimization* of programs written in SELF, a pure object-oriented programming language designed for rapid prototyping of code, which provides expected behavior debugging. As implemented, the system performs run-time compilation of SELF code, using optimized procedures when possible, and re-compiling unoptimized procedures when those procedures are being debugged (after debugging, the optimized versions can replace the unoptimized ones). Similar techniques could be used with a "fat" binary executable that contains two versions of all procedures: one version optimized and the other version unoptimized.

The techniques outlined by Hölzle et al. [1992] are applicable to many common scalar optimizations, including dead code elimination, strength reduction, global common-subexpression elimination (CSE), loop unrolling, and code hoisting. Interesting object-oriented optimizations that can be dynamically undone in SELF include *inlining* of methods, *customization* (multiple versions of polymorphic object methods are produced by the compiler, each one customized for a particular type, allowing static binding and inlining of otherwise dynamically-dispatched procedure calls), and *splitting* (a similar customization of expressions for specific types).

Hölzle et al. specifically restrict dead store elimination and tail recursion optimizations. More generally, however, optimizations are unrestricted only between well-defined *interrupt points* occurring in method prologues and at the end of loop bodies. Global optimizations (or local optimizations that cross *interrupt points*) are not allowed.

In contrast to the work of Zellweger [1984], discussed below, the SELF debugger allows for generalized single stepping and the ability to continue execution until the end of a procedure (the *finish* debugger command). Asynchronous breakpoints, however, are delayed until the next *interrupt point.* Setting breakpoints involves code modification (a call to the debugger is inserted) and recompilation of the method.

For his LOIPE system, Feiler [1982] uses transparent incremental recompilation for debugging when a user sets a breakpoint, but, in contrast to Hölzle et al., does not dynamically recompile procedures that are currently executing. Instead, users can adjust the level of initial optimization.

Dynamic recompilation (or "deoptimization") solves a few of the problems of using unoptimized code for debugging: the recompilation is transparent to the user, and is selective, allowing the rest of the program to execute at full speed. However, other problems are not solved: the optimized and unoptimized code may still behave differently, and this technique is not applicable when the optimizations must be applied to ensure the program will execute.

2.5.4 Detection and Recovery of Noncurrent Variables

Hennessy [1982] introduces the notion of *noncurrent* variables: "variables whose values do not correspond to those in the original program". Variables may be noncurrent at a particular point in a program because optimizations have caused a value to be assigned too early (e.g., code hoisting) or to be obsolete (e.g., dead store removal).

Hennessy provides algorithms for identifying noncurrent variables, and suggests modifications to the expression DAG (directed, acyclic graph) to allow reconstruction of the values that the variables should have in the unoptimized version of the program. Several logical errors in these algorithms are corrected by Wall et al. [1985]. Although some of Hennessy's work may be extended to global optimizations, his main emphasis is on local optimizations. Copperman and McDowell [1993] extend Hennessy's algorithms and provide a brief review of contemporary attempts to solve the currency problem in general.

2.5.4.1 Generalizing the Currency Problem

Copperman [1993a,b, 1994] concentrates on providing expected behavior debugging for a wide range of optimizations. His

> ...work is applicable in the presence of any sequential optimizations that either do not modify the flow graph of the program or modify the flow graph in a constrained manner. Blocks may be added, deleted, coalesced, or copied; edges may be deleted, but control flow may not be radically changed. [Copperman 1993b, p. 5]

Examples of optimizations that are not supported include loop interchange and replacement of a bubblesort by a quicksort routine. Again, the supported optimizations are similar to the common scalar optimizations shown in Figure 2.1, and the more aggressive vectorizing and flattening optimizations are not supported.

Zellweger [1984] implements a debugger that provides expected behavior for inline procedure expansion and cross-jumping. The debugger makes use of static information generated at compile time, and (only when necessary) dynamic information generated during execution (via the use of instrumentation that could change the run-time characteristics of the program).

Adl-Tabatabai and Gross [1994] examine how global optimizations effect variable currency (the data-value problem). They take advantage of invariants preserved by correct optimizations for code hoisting and dead code elimination. Their approach is restricted to transformations that do not perform arbitrary code movement and elimination.

In his dissertation, Adl-Tabatabai [1996] presents a detailed analysis of the problems caused by scalar optimizations and shows how to track the effects of these optimizations so that expected behavior debugging can be provided. He implements these techniques for the `cmcc` compiler, a retargetable optimizing C compiler.

Adl-Tabatabai's methods can handle common scalar optimizations such as those shown in Figure 2.1. However, he does not consider loop transformations that are applicable to parallelization or optimizations which improve memory use. Because these transformations occur at such a high level of abstraction, he suggests that "the best approach to handling loop transformations may be to expose these optimizations to the user by rewriting the source to reflect the effects of loop transformations" [Adl-Tabatabai 1996, p. 163].

Generally, these systems require that some extra notations be added to the transformations so that debugging can be performed. In the case of Zellweger, debugging support for only two optimizations is provided, and each optimization requires significant coding to provide the debugging. In the case of Adl-Tabatabai [1996], the compiler transformations must annotate the IR with special markers and attributes that depend on how the transformation changes the program. Sometimes the annotation depends on an understanding of the semantics of the programming language.

In contrast, the debugging algorithms described in this dissertation (Chapter 3), as implemented in the KHEPERA system (Chapter 4), track transformations based only on low-level changes made to the AST—without any knowledge of program semantics. The advantage of this approach is that the tracking is performed transparently, without special aid from the transformation writer. The KHEPERA debugging system is ready as soon as the transformations are written, and can be used immediately to debug them. If desired, the implementor can later add special per-transformation annotations to the AST and implement, for example, Adl-Tabatabai's algorithms on top of the basic KHEPERA debugging system (this example is discussed in Chapter 5).

2.5.4.2 Variable Ranges

As mentioned earlier, typical debuggers use tables that maintain a one-to-one correspondence between source variables and memory locations. Coutant et al. [1988] implemented a system for HP9000 Series 800 RISC-based compilers that maintains range data for variables. For each variable, a map exists from machine code addresses (i.e., locations in the program) to memory locations, registers, or constant values. When the value of a variable is requested, the address of the current breakpoint is searched for in the table, and, if found, the current location of the variable is used to display its value. If the variable has been subjected to constant folding, then a constant is stored in the table. If the address is not found in the table, then the variable is not current and this information is given to the user. No attempt to reconstruct the value of noncurrent variables is made.

This technique concentrates on data-value problems. Statement boundaries are tracked by labeling the first instruction associated with the statement. When statements are moved or deleted, the label is moved to the next instruction.

These techniques solve the stated goal of "tracking the locations of a variable's values from memory through registers", and appear to work well for low-level optimizations. The set of optimizations and transformations discussed include: copy elimination, register allocation, register spills, and instruction scheduling. Loop variables that were eliminated due to strength reduction and induction variable elaboration can be recreated by the debugger. Higher-level or aggressive global optimizations, however, cannot be handled by the techniques presented.

2.5.5 Debugging Parallelized Programs

2.5.5.1 Dynamic Order Restoration and Structural Mapping

Cohn [1992] describes a theoretical framework for expected behavior debugging of the execution of single address space, sequential programs on a distributed memory MIMD machine.

Cohn identifies two requirements for expected behavior debugging of parallelized code: dynamic order restoration and structural mapping. Dynamic order restoration reproduces the sequential ordering of the unparallelized program. Structural mapping creates a map from variables in the unparallelized program to variables in the parallelized program (e.g., an array in the unparallelized program is spread over n processors in the parallel version). These functions allow a debugger to provide a "sequential view of parallel execution" [Cohn 1992, p. 1].

Cohn requires the computations performed by the parallelized code to be implemented in the same order as the computations performed in the sequential code (for example, the replacement of a sequential reduction by a parallel reduction may make it impossible to construct a debugger for the program [Cohn 1992, p. 23]). Because of this restriction, Cohn's work is not readily applicable to languages that contain explicit parallelism or to aggressive parallelizing compilers that perform complicated transformations, such as the flattening of nested-data parallelism.

The KHEPERA tracking algorithms discussed in this dissertation (see Chapter 4) could be used to directly extend Cohn's work by providing support for the complicated structural mapping required in the face of aggressive program transformations.

As with other expected behavior debugging schemes, Cohn's debugger may require a program to be rerun from the beginning or to be run without full optimization (which, in this case, means with reduced parallelism) [Cohn 1992, p.54]. Since Cohn assumes that the parallelizing compiler is correct, his debugger does not have to be able to debug the newly added interprocessor communications primitives (these communications primitives are hidden from the user's view).

Cohn uses a 4-line matrix multiply program to demonstrate the transformations necessary to debug after block and cyclic distributions have been performed.

2.5.5.2 Instant Replay

LeBlanc and Mellor-Crummey [1987] propose a method of saving the "relative order of significant events as they occur, not the data associated with such events". The saved order of events is used to reproduce the execution behavior of parallel programs.

The idea of saving and replaying program state was probably first proposed by Balzer [1969] for debugging serial codes. Balzer's system stored a large amount of program history, and then provided an environment for writing routines to query the *history tape*, thereby answering questions necessary for debugging the program.

2.5.5.3 Instrumentation

Gupta [1988] presents a technique that integrates the debugger with a trace scheduling compiler for highly parallel VLIW machines. In this system, the programmer specifies monitor points and the compiler instruments the original code with the monitor. This process is faster than recompiling the whole program, since the semantics of the program do not change. However, instruction scheduling can be perturbed by the instrumentation, so the program being debugged is not exactly the program that will be executed if all debugging is removed.

2.5.5.4 Reverse Execution

Automatic instrumentation of code is also proposed by Tolmach and Appel [1991] for the debugging of Standard ML, a general purpose programming language with first-class functions, strong typing, and polymorphism [Milner et al. 1997]. This system supports data value discovery via *reverse execution*, which is implemented using a combination of checkpointing and re-execution. The debugger is implemented in ML entirely within the ML concurrency model.

2.5.5.5 Other Viewpoints

Fritzson [1983] describes an integrated programming environment that relies on incremental compilation for debugging programs. Optimizations are restricted so that they do not span statement boundaries.

Pineo and Soffa [1991] present a technique of *global renaming* which enables the debugging of transformed and parallelized sequential programs. This converts the program into a single-assignment form, allowing the debugger to work on the parallelized version of the code, while presenting a sequential-source viewpoint of data values to the programmer. Variable values are not available when code is moved forward ahead of the current breakpoint, and code location problems are ignored.

2.5.6 Summary of Expected Behavior Debugging

Expected behavior systems devote a great deal of effort to hiding the effects of optimizations and to discovering when these effects cannot be hidden. The set of acceptable optimizations for which expected behavior debugging can be provided is small when compared with modern optimization techniques. For these modern systems, a single method that allows for debugging the transformation system and user programs is necessary: the overhead in implementing a single debugging system is often too large for a research or prototyping environment.

2.6 Truthful Behavior Debugging of Optimized Programs

2.6.1 Non-Graphical Debugging

The FDS debugger [Warren, Jr. and Schlaeppi 1978] appears to be one of the first systems that attempted to provide truthful debugging in the face of optimization. The system concentrates on identification (and recovery) of noncurrent variables in the face of some simple scalar optimizations, reporting when a variable use was deleted or moved. The compiler/debugger system includes a "no source change" optimization mode, which prevents the compiler from eliminating variable uses, but which allows CSE within a single statement and full optimization of compiler-generated temporaries. The proposed implementation of this system relies on one-to-one maps from a location in the original input program to a location in the generated output code (microcode, in this case)—no attempt is made to report which optimizations cause the changes.

2.6.2 Selective Highlighting

The CONVEX[1] CXdb debugger [Brooks et al. 1992; Streepy, Jr. 1994] provides truthful behavior symbolic debugging in the face of aggressive optimizations. The compiler provides detailed mappings between source statements and object code, enabling a visual debugger to highlight interesting portions of the original source code during the debugging process.

This approach is excellent for simple code motion, dead store elimination, CSE, and code sharing. However, when more complicated optimizations are performed, the highlighted section of code may not be helpful to the programmer. For example, the composition of loop inversion and loop reversal can still be visualized using the highlighting technique. However, since the highlighting is performed on the original source code, the portions of the program which are highlighted may span several lines and may change radically while single stepping the program, confusing the user or providing little understandable information.

The approach of Edelstein et al. [1992] is similar to that of Brooks et al.: a mapping between source statements and object code is maintained and is used to

[1] CONVEX is a trademark of Convex Computer Corporation.

provide truthful debugging with a visual interface.

2.6.3 Exposing the Transformation Process

One early transformational software system [Kuck et al. 1981] allowed the user to manually display a snapshot of a program during the transformation process. These displays, however, were designed to help the user understand the optimization process so that other optimizations or transformations could be (manually) applied. This work was not directed toward the understanding of the transformation process in the context of debugging and is discussed in the next section along with other early transformational programming systems.

Cool [1992] proposes, but does not implement, a system that explains software pipelining and loop unrolling by displaying original source code and transformed source code side-by-side in a window. No provisions are made for stepping through transformations, and it is unknown how well this approach would work in the face of more complicated transformations, especially when several are composed together on the same few lines of code.

2.6.4 Summary of Truthful Behavior Debugging

Since expected behavior debugging is difficult or impossible to provide for many optimizations [Brooks et al. 1992], it is not suitable for a general method for implementing debugging support. Expected behavior debugging may also require optimization-specific support in either the compiler or the debugger [Zellweger 1984], thereby increasing the complexity of the debugging support that must be provided by the language implementer.

When compilers for novel languages are considered, expected behavior (even if it could be provided in all cases) would most likely make compiler debugging a very difficult task:

1. Expected behavior would hide the implementation errors that led to the need for debugging the compiler.

2. The reverse transformations necessitated by expected behavior might have been implemented incorrectly.

In this case, truthful behavior would be superior, especially if the proper information about the transformations could be provided.

Users of novel compilers, especially those targeted at high-performance computers, may want to see exactly how their program was transformed so they can adjust their code for optimum performance (this interest has traditionally been demonstrated among scientists using FORTRAN compilers on supercomputers: although the scientists are not compiler implementors, they care very much how and in what ways their programs are being optimized—I have already observed this behavior in sophisticated end-users of the KHEPERA system who were not directly involved in the implementation of KHEPERA). Again, truthful behavior is desirable for these users.

The main problems with current truthful debugging systems are:

- The visual interfaces provided, which highlight the original source code, or which show before and after views of transformed source code, are best used when single-stepping through the program, and may become extremely confusing when debugging code after complicated loop transformations have been applied.

- Changes to the transforms must be made so that debugging can be supported (compared with expected behavior systems, fewer changes are needed for truthful debugging systems, but any additional work beyond writing the transformations themselves increases implementor overhead and the chances for programmer error—and may result in a decision not to implement debugging support).

I believe the ability to view intermediate forms of the transformed program, either as an AST or as source-code in some extended language, will be more helpful than just seeing the original source program. The ability to view these intermediate transforms will provide valuable information to the transformation implementor and to the dedicated programmer. Further, the ability to view the transformations at different levels of abstraction (e.g., as a single transformation, or as a related set of transformations) will make the examination of many thousands of tiny transformations manageable for the human (in our PROTEUS-to-C translator, the translation of a simple quicksort [Cormen et al. 1991, Chapter 8] program requires more than 5000 transformation applications).

When the optimizations being applied are common scalar optimizations, the information tracked by the system described in this dissertation can be used to augment existing debugging systems or future expected behavior systems (e.g., using techniques described in [Zellweger 1984; Copperman 1993a; Adl-Tabatabai 1996]). This augmentation can provide expected behavior debugging for scalar code, with truthful behavior fallback when more complicated vectorizing or parallelizing transformations are involved. Significantly, the support for truthful behavior debugging can be provided in a manner that is transparent to the transformation writer—all of the necessary support can be encapsulated in the low-level transformation engine.

2.7 Debugging Transformation Systems

The work on debugging transformational compilation systems is sparse. Loveman [1977] presents one of the first papers suggesting that source-to-source transformations "...provide a coherent model of the compilation process for high level languages and for much of code generation". In Loveman's early transformational compilation system, there appear to be utilities available to print and manipulate the intermediate representation, but there does not appear to be a way to navigate through a hierarchy of applied source-to-source transformations. Similarly, the system described by Kuck et al. [1981] has the ability to "regenerate a source program" after each transformational module has been (manually) applied. However, there is no provision for navigating through snapshots of these source programs, or for posing or answering debugging queries.

Partsch and Steinbrüggen [1983] present an excellent overview of program transformation systems (Partsch [1990] presents a more recent overview), but debuggability is never a driving issue. In general, the composition of language translators and the subsequent need to compose debuggers is not explored in this literature.

One powerful transformation system, REFINE[2], contains a complete, general purpose programming language built on a LISP system [Reasoning Systems 1990]. KIDS, the Kestrel Interactive Development System, is a tool which runs in the REFINE environment and provides, among many other features, the ability to examine sequential snapshots from a program transformation sequence [Smith 1990]. However, KIDS does not appear to provide any support for composing its information with a pre-existing debugger, and it is unclear if sufficient information is even available to support this level of debugging.

Bertot [1991] outlines a method of *subject tracking* for debugging interpreters based on term rewriting of the λ-calculus. Tip [1995] defines a more general system of *object tracking* that is also used for debugging inside term rewriting systems. His

[2] REFINE is a trademark of Reasoning Systems, Inc.

basic algorithm is to annotate the initial term and propagate origins during rewriting. Both of these methods can be automated, so the author of the term rewriting rules obtains transparent debuggability of the rewriting system. In this way, this work is similar to KHEPERA's use of tracking to debug the translator. However, both systems are restricted to interpretive languages implemented using a term rewriting system: there is no attempt to provide answers to debugging questions that would be useful if the system were composed with another compiler/debugger combination, and there is no attempt to provide navigation forward *and* backward in the term rewriting sequence.

Since these systems track terms, they are unable to track through a rewrite rule that changes the term, even though the semantics are preserved. For example, the following rewrite rule:

$$\texttt{trans(plus(}E_1\texttt{,}E_2\texttt{))} \rightarrow \texttt{seq(trans(}E_1\texttt{),seq(trans(}E_2\texttt{),add))}$$

the tracking system is unable to track the `plus` to the `add`. In contrast, KHEPERA-style tracking would either track the `plus` to the `add` or track the `plus` to the outermost `seq`, depending on how the transformation rule was written.

2.8 Conclusion

Hennessy [1982] did early work on the problem of providing expected behavior symbolic debugging in the face of aggressive scalar optimizations. Theoretical and practical considerations of this goal have been advanced significantly by Zellweger [1984], Copperman and McDowell [1993], and Adl-Tabatabai [1996]. Unfortunately, the techniques for providing expected behavior debugging are complicated and have not yet found their way into widespread use in commercial debuggers. Also, during this time, compilation techniques have changed dramatically, and there is now significant interest in parallelization, vectorization, and aggressive transformational optimizations that are not be handled well by the expected behavior techniques currently proposed.

The KHEPERA system does not provide expected value behavior, but has several advantages over systems which do:

1. Debugging information is tracked transparently at a low level of AST manipulation. When writing a transformation rule, the implementor does not have to make special provisions for the debugging system.

2. Debugging information is tracked in the compiler, and does not require any changes to the executable or restrictions in the type or complexity of transformations performed.

In my work, I have concentrated on a framework for symbolic debugging in the face of aggressive non-scalar optimizations. This framework is targeted at high-level language processors that are composed with existing compilers. The solution space does not allow instrumentation, dynamic recompilation, or any other methods which could perturb the run-time characteristics of the program being debugged. Further, the solution requires minimal assistance from the transformation implementor and does not restrict the type or complexity of transformations performed.

In the next chapter, I will provide an overview of the problem and solution spaces being explored, and in Chapter 4, I will discuss an example implementation of the proposed techniques.

Chapter 3

Tracking Algorithms

3.1 Introduction

This chapter provides an overview of the problem and the desired solution techniques, using examples from the literature for illustration. The transformational view of a language processor is formalized and the basic elements of transformations (e.g., copy, delete) are described. Each of these elements causes specific information about the ongoing AST transformation to be tracked. This information can be used to answer specific questions that are necessary to provide transformation replay and debugging of the transformations and of end-user codes.

3.2 Overview

3.2.1 Problem

Expected behavior debugging of optimized code is viable only for a relatively small set of scalar optimizations. As optimizations become more complicated or are aggressively composed, the ability to provide expected behavior debugging becomes increasingly difficult. When vectorizing, parallelizing, or flattening optimizations are considered, expected behavior debugging, even if possible, may not be helpful to the end-user.

Expected behavior debugging is seldom useful for the compiler implementor who requires detailed knowledge of optimization application during the debugging process. Further, providing expected behavior debugging greatly increases the complexity of implementing a compiler and debugging system. In practice, commercial debuggers do not consistently provide expected behavior [Copperman and McDowell 1993], either because implementing expected behavior is too expensive or because programmers do not give a high priority to expected behavior.

In contrast, Convex has created a commercial debugger [Brooks et al. 1992; Streepy, Jr. 1994] that provides a truthful debugging environment. The Convex debugger, however, displays all debugging information in terms of the original source program using highlighting and graphical annotations. For complicated or composed optimizations, the original source program cannot capture the essence of the optimizations and convey that information to the user. As a partial solution to this problem, Loveman [1977] suggested using views of the partially transformed program as a means of explaining optimizations, but his work was not concerned with debugging issues or with navigation between views of partially transformed programs.

Since many research and conventional compilers are written as language processors using a transformational viewpoint [Pittman and Peters 1992; Cordy and Carmichael 1993; Appel 1997], the problem of providing truthful debugging with useful explanations is explored in this dissertation within the framework of a transformational programming system. The debugging capabilities and restrictions on debugging methods will be discussed in the next two sections.

3.2.2 Desired Behavior

The debugging method described here, an example of which is implemented in the KHEPERA system (see Chapter 4), concentrates on the ability to handle a large class of transformations while providing truthful behavior (see Section 2.4) and while providing the ability to debug the transformation system itself (see Section 2.7). This system provides:

- The ability to navigate through intermediate versions of the transformed program.

- The ability to navigate through the transformations at multiple levels of transformation abstraction. For example, the user can select a single transformation and see how that transformation was applied to the AST, or the user can select a set of transformations that collectively provide some abstract transformation (e.g., CSE) and see the effects of applying that set of transformations to the AST.

- The ability to provide specific services that support the debugging of the final transformed output (see Section 2.1):
  - setting breakpoints,
  - determining current execution location (e.g., in response to a breakpoint or program exception),
  - reporting a procedure traceback, and
  - displaying values of variables.

3.2.3 Desired Solution Techniques

The goal of this dissertation is to provide effective truthful debugging of the production version of the optimized program—without changes that would impact the run-time characteristics of the program. This implies that the solution presented here cannot use any of the more invasive methods for providing debuggability:

- The compiler may not limit the optimizations performed when debugging is allowed.
- The compiler may not insert any special instructions into the transformed program for debugging support (i.e., the compiler will not instrument the program in any way).
- The debugger may not dynamically recompile portions of the program.
- The debugger may not insert "silent breakpoints" or other instrumentation into the program at run time (i.e., the only breakpoints that will affect program execution are those inserted with the full knowledge and consent of the user).

Given these restrictions, which prohibit debug-time changes to the program, the solution presented here must rely on:

1. debugging information computed during the compilation/translation phase, and
2. information available to a standard (i.e., non-invasive) debugger (e.g., the program counter and call stack are available when the program encounters a user-defined breakpoint or a run-time exception).

```
10 int f(int x) {
11     return x + x;
12 }

20 int g(int y, int z) {
21     int a = f(y);
22     int b = f(z);
23     return a + b;
24 }
```

$\longrightarrow$

```
10 int f(int x) {
11     return x + x;
12 }

20 int g(int y) {
21     int a = y + y;
22     int b = z + z;
23     return a + b;
24 }
```

Figure 3.1: Procedure Inlining Transformation

3.2.4 Code Location Problems

Previous solutions to examples of code location problems from the literature on symbolic debugging of optimized code were discussed in Section 2.3.2 (page 12). In this section, these examples will be used to explore the solution proposed in this dissertation. At times, the restrictions listed above will be too severe to provide the exact information that other techniques have provided. In these cases, a slight relaxation of the restrictions (e.g., permitting silent breakpoints) may be sufficient to provide the same level of information provided by other techniques. However, debugging systems can always be improved by collecting additional information and implementing algorithms to handle specific cases. The especially interesting and novel component of the debugging techniques proposed in this dissertation is that valuable debugging information can be provided *independent of the transformations*. Special transformation-specific debugging information can be added to the compilation system, but is not required. This allows experimental transformations to be added and tested quickly—without breaking the debugging system until the transformation is enhanced with the appropriate debugging information.

3.2.4.1 Inline procedure expansion

The procedure inlining example from Section 2.3.2 (page 12) is shown in Figure 3.1. The KHEPERA system would use the tracking algorithms to map syntactic elements on line 11 (i.e., x or $+$) to corresponding syntactic elements on lines 11, 21, or 22.

If KHEPERA is composed with a standard debugger, such as `dbx` or `gdb`, the information derived from tracking could be used to set breakpoints at line granularity,

```
30 if (a == b) {
31     x = 1;
32     y = 2;
33     z = 3;
34 } else {
35     x = 2;
36     y = 2;
37     z = 3;
38 }
```

$\longrightarrow$

```
30 if (a == b) {
31     x = 1;
34 } else {
35     x = 2;
38 }
39 y = 2;
40 z = 3;
```

Figure 3.2: Cross-Jumping Transformation

since this is the only granularity supported by the underlying debugger. However, if a debugger with more capabilities is available, then a finer breakpoint granularity will be possible. The HP[1]/DDE Debugger (`dde`) can set a breakpoint on an individual statement, even if more than one statement appears on a single line of source code. The CONVEX Visual Debugger (`cxdb`) provides even finer granularity, allowing breakpoints to be set on expressions *within* statements. If KHEPERA is composed with such a debugger, then the information from the tracking system can be used to set breakpoints at a finer granularity.

3.2.4.2 Cross-jumping

The cross-jumping example from Section 2.3.2 (page 12) is shown in Figure 3.2. In this example, the user set a breakpoint on line 32 or line 36 in the original source.

In contrast to Zellweger's silent breakpoint solution, KHEPERA would either set the breakpoint on line 30 or 39, depending on how the transformation was written. For example, say the transformation was written in the most naïve way possible, simply matching the **if** statement on line 30, copying the tail from the **then** part, and deleting the tail in the **else** part. With this transformation, a breakpoint set on line 32 would be tracked to line 39, but a breakpoint set on line 36 would be tracked to line 30 (since line 36 was deleted, tracking uses the place where the transformation rule matched). If the transformation implementor wrote a slightly more complicated transformation rule, however, there would be a notation in the tracking information that lines 32 *and* 36 are were copied to line 39, and that lines 33 *and* 37 were copied

[1]HP is a trademark of Hewlett-Packard Corporation.

to line 40. In this case, setting the breakpoint on either line 32 or line 36 would result in a breakpoint set on line 39.

As will be shown in later examples, the exact behavior of KHEPERA is dependent on how the implementor wrote the transformations being tracked. If the implementor takes absolutely no care when writing the transformation, then KHEPERA will provide less precise information. With a small amount of care, the information provided by KHEPERA can be much more precise. Note, though, that the overhead required from the transformation implementor is still very small compared with the overhead involved in implementing expected behavior debugging algorithms.

Since the tracking functions allow the transformation application to be unwound and viewed at any intermediate tree, the debugging system can also show the user the transformed code and allow the user to fine-tune the placement of the breakpoint in terms of the transformed output. This solution has several advantages:

- the debugger doesn't insert any silent breakpoints (which might unexpectedly change the run-time characteristics of the program, especially if the silent breakpoints are set inside an inner loop),
- the debugger displays precise information about the exact placement of the breakpoint, avoiding surprises such as those that occur when code motion causes a breakpoint intended to be inside a loop to be placed outside the loop,
- the user views the breakpoint in the local context of the transformed program, and can adjust the position of the breakpoint using new information obtained from this view, and
- the sophisticated end-user or transformation implementor can also adjust the breakpoint in partially transformed views of the program (this might be especially helpful when the transformations terminate with machine code generation: viewing and setting breakpoint positions in the final transformed version of the high-level code might be more helpful than doing so in the machine code version).

3.2.4.3 Procedure Inlining Together With Cross-jumping

When both procedure inlining and cross-jumping are combined, the results can require a many-to-many mapping. As shown in Figure 3.3, a breakpoint set in the original code at *either* line 43 *or* line 46 will require breakpoints to be set in the transformed code at lines 50, 66, *and* 72. Depending on how the transformations were implemented, KHEPERA would map a breakpoint on line 43 in the original source code to lines 41, 61, and 67 or to lines 50, 66, and 72.

3.2.5 Data-Value Problems

Several examples of data-value problems were outlined in Section 2.3.3 (page 15).

For expected value debugging, one of the biggest problems is to determine variable currency and to recompute expected values for noncurrent variables. The typical use of currency determination algorithms occurs when a variable value is requested at a specific breakpoint. Since the debug tracking described in this dissertation maps syntactic elements in the original source program to syntactic elements throughout the transformation process, semantic questions, such as those about the variable currency at a breakpoint, cannot be answered using the tracking machinery. Instead, typical questions that can be answered are:

- Given an assignment to a variable in the input source, where is the equivalent assignment in the transformed source?
- Given a use of a variable in the input source, where is the equivalent use in the transformed source?

The algorithms necessary to answer these questions are identical to the algorithms used to answer questions about breakpoints.

The discussion of location problems has provided an introduction to the capabilities of the tracking system, and the full discussion of variable value determination and the data-value problem is deferred until Section 5.2.3 (page 123). Below, a more formal view of the transformation process is presented and the details of the low-level tracking are explained. Chapter 4 will show how these tracking algorithms are actually implemented in an example transformation system, and Chapter 5 will discuss various applications that can be built using the KHEPERA system.

3.3 Formalism of Structure-Changing Program Transformation

An AST T is a pair (N, E), where N is a finite set of nodes, and E is a set of edges, forming a rooted tree. Children are ordered from left to right under the parent node.

```
38 int b;

39 int f(int x) {
40     int a;
41     if (x == 3) {
42         a = 1;
43         b = 2;
44     } else {
45         a = 2;
46         b = 2;
47     }
51     return a;
52 }

60 int g(int c) {
80     int n = f(c);
81     int m = f(d);
82     return n + m;
83 }
```

⟶

```
38 int b;

39 int f(int x) {
40     int a;
41     if (x == 3) {
42         a = 1;
44     } else {
45         a = 2;
47     }
50     b = 2;
51     return a;
52 }

60 int g(int c) {
61     if (c == 3) {
62         n = 1;
63     } else {
64         n = 2;
65     }
66     b = 2;
67     if (d == 3) {
68         m = 1;
69     } else {
70         m = 2;
71     }
72     b = 2;
82     return n + m;
83 }
```

Figure 3.3: Procedure Inlining and Cross-Jumping Transformations

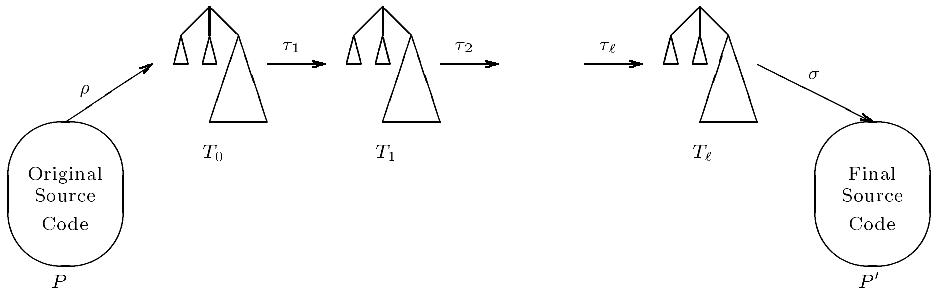


Figure 3.4: Transformation Process

P is a syntactically and semantically well-formed program in the input language, L, and T_0 is an AST created by parsing P during the parsing step, ρ. T_ℓ is the final transformed AST, and P' is a valid program, constructed from T_ℓ, in the output language, L'. P' is constructed by "unparsing" the AST in the σ step.

The transformation process is viewed as a sequential application of various transformation functions, $k = 0, \ldots, \ell, \tau_{k+1}(T_k) = T_{k+1}$, to the ASTs, as shown in Figure 3.4.

Although not all language processors are implemented in this way, the work presented here assumes this model since it can be used to implement a wide variety of language processors, and this model is in widespread use in the research compiler and domain-specific language communities. Recent textbooks on general compiler design [Pittman and Peters 1992; Appel 1997; Muchnick 1997] have also advocated the transformational approach for compiler implementation.

3.3.1 A Tree Transformation Library

An AST $T_k = (N, E)$ is transformed into a new AST $T_{k+1} = (N', E')$ by the application of a transformation function τ. This transformation function matches some subtree, σ_m, rooted at m in T_k and performs some sequence of operations on T_k. A subtree σ_m contains m and all of the descendants of m, if any descendants exist. A match identifies some subset of σ_m. τ can perform the following operations:

1. Update attributes on one or more nodes. Attributes are programmer-defined values which are associated with a node. For example, an `Integer` node may have an attribute which contains a 32-bit integer value.

2. Add nodes to the tree.

3. Delete nodes from the tree.

A typical tree transformation library or system will support low-level functions which will support the following basic abstractions:

Node create. Create a new node.

Node destroy. Destroy an existing node.

Node copy. Create a new node n_1, and copy to n_1 all of the attributes from an existing node n_0.

Subtree replacement. Replace a subtree, σ_m, with a new subtree, σ_n.

The library will also have other functions that insert new subtrees into the AST, that disconnect and delete existing subtrees from the AST, and that perform other abstract operations on the tree. However, for the purposes of tracking, only the four low-level functions outlined above are of interest: node create, node destroy, node copy, and subtree replacement. For example, consider an example of constant propagation and constant folding shown in Figure 3.5.

τ_1 first matches a subtree consisting of a single identifier that was assigned a constant in a previous statement (b_2), replaces the identifier (b_2) with a copy of the constant node (2_1 is copied to 2_2), and deletes the subtree containing the previous assignment ($\sigma_{=_1}$). An actual rule for constant propagation must perform more complicated analysis, but this description is sufficient for this example.

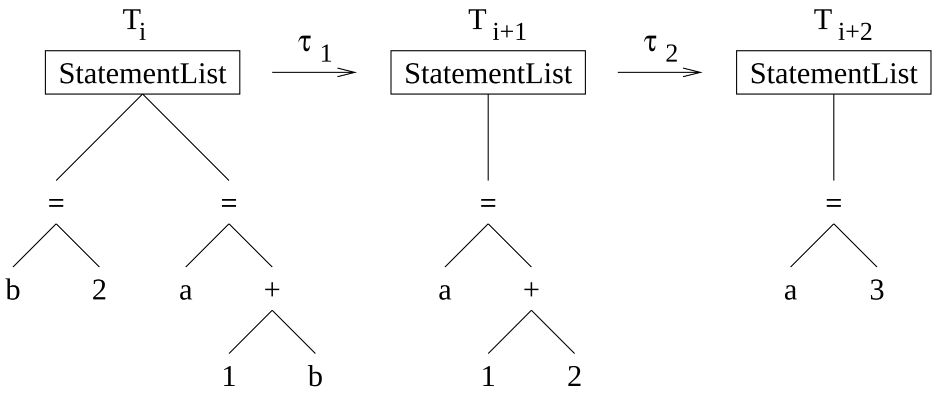
τ_2 matches a subtree rooted at an addition node with two integer children (σ_{+_2}, 1_2, and 2_2) and replaces the matched subtree with a new integer node (3_2), having an attribute that is the sum of the values of the two integer children.

3.3.2 Tracking the Transformations

The goals of the tracking system are to provide tracking of debugging information that is:

1. independent of the transformation being performed, and

2. transparent to the transformation implementor.



T_i

```
1) b₁ =₁ 2₁
2) a₂ =₂ 1₂ +₂ b₂
```

$\xrightarrow{\tau_1}$

T_{i+1}

```

2) a₂ =₂ 1₂ +₂ 2₂
```

$\xrightarrow{\tau_2}$

T_{i+2}

```

2) a₂ =₂ 3₂
```

Figure 3.5: Constant Propagation and Constant Folding

These two goals can be attained if the tracking takes place at a very low level in the transformation engine: at the level of node and subtree creation and destruction. This way, the tracking is performed transparently whenever the transformations are applied, and the author of the transformation does not have to make any special effort to guarantee that tracking is performed.

For sufficient data to be collected at the lowest level of the transformation engine, the following assumptions are made:

- the index i of the current AST T_i is available;
- the current transformation τ is available (τ is the transformation which is applied to T_i to generate T_{i+1});
- the subtree σ_m, at which τ matched T_i, is available; and
- information relating the current transformation, τ, to other semantically-related transformations is available.

These assumptions are reasonable since this is a small amount of data that can be made, at minimum, globally available within an implementation of the transformation engine. Note that for a given tree T_i, a transformation τ can only match at *one* subtree—subsequent matches of the same rule are performed on subsequent subtrees.

Given this information, *tracking* is performed by logging events to a database. This database can be implemented as a flat file containing the tuples described below, or as some more sophisticated data structure. Since matching and manipulating the AST can be viewed at several levels of abstraction, changes to the AST can be tracked in several ways:

1. A transformation function τ matches a subtree σ_m and transforms the AST T_k into the AST T_{k+1}. This can be denoted by the following tuple:

   $$(k, \tau, \sigma_m) \tag{3.1}$$

   The type signature for this tuple is:

   $$\text{tree index} \times \text{rule} \times \text{subtree}$$

2. The transformation function τ changes the AST by manipulating subtrees. A specific subtree can be deleted, copied, or replaced. For the algorithms described later in this chapter, replacement is the only low-level operation on subtrees that must be tracked:

   $$(k, \tau, \sigma_m, \text{replace}, \sigma_{\text{old}}, \sigma_{\text{new}}) \tag{3.2}$$

   The type signature for this tuple is:

   $$\text{tree index} \times \text{rule} \times \text{subtree} \times \text{“replace”} \times \text{subtree} \times \text{subtree}$$

3. Ultimately, the functions that operate on subtrees modify individual nodes. A node can be created, deleted, or copied. These operations can be denoted by the following tuples:

   $$(k, \tau, \sigma_m, \text{create}, n_{\text{new}}) \tag{3.3}$$

   $$(k, \tau, \sigma_m, \text{delete}, n_{\text{old}}) \tag{3.4}$$

   $$(k, \tau, \sigma_m, \text{copy}, (n_{\text{old}_1}, n_{\text{old}_2}, \ldots, n_{\text{old}_c}), n_{\text{new}}) \tag{3.5}$$

The type signatures for these tuples are:

$$\text{tree index} \times \text{rule} \times \text{subtree} \times \text{“create”} \times \text{node}$$
$$\text{tree index} \times \text{rule} \times \text{subtree} \times \text{“delete”} \times \text{node}$$
$$\text{tree index} \times \text{rule} \times \text{subtree} \times \text{“copy”} \times \text{list of nodes} \times \text{node}$$

In addition to the ability to track nodes and subtrees, more abstract tracking is also possible. For example, a high-level compiler optimization, such as constant folding, may be implemented using a set of transformation functions, $\{\tau_1, \tau_2, \ldots, \tau_n\}$, which should be considered together. For the rest of this section, however, only the lower-level abstractions will be discussed, since these abstractions provide a foundation for any additional tracking that is performed.

The tuples described above can be collected as the AST is undergoing transformation, and can be associated with the nodes that are affected by the changes described by the tuple. For example, the tuple (k, τ, σ_m) affects all of the nodes in T_k and T_{k+1}, whereas the tuple $(k, \tau, \sigma_m, \text{replace}, \sigma_{\text{old}}, \sigma_{\text{new}})$ tuple affects only the nodes in the σ_{old} and σ_{new} subtrees.

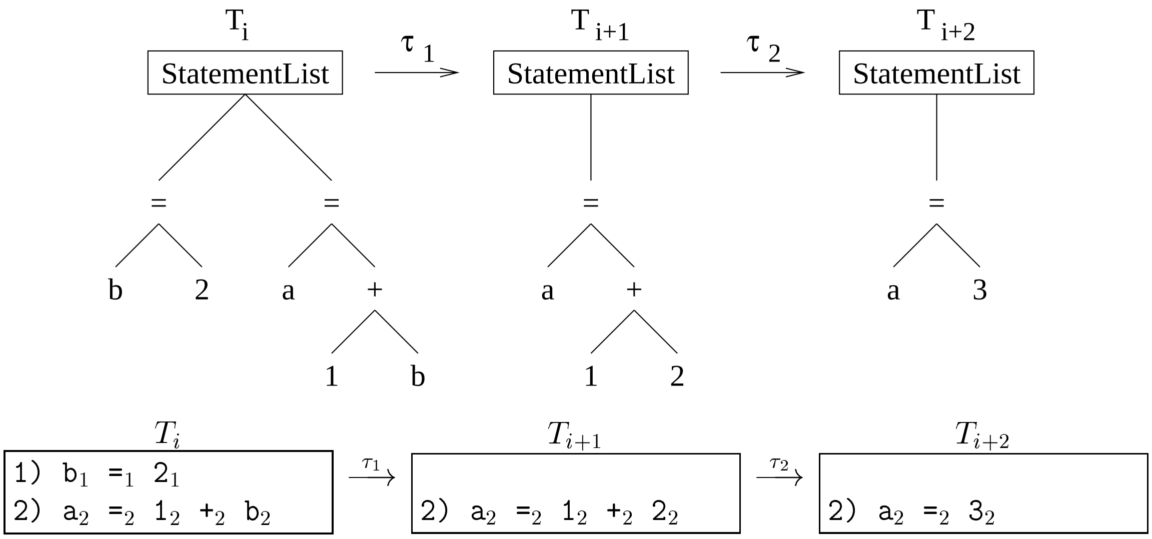
Continuing the example from Figure 3.5, the transformations would cause the tuples shows in Figure 3.6 to be logged. Examples in Section 3.4.3 and Section 3.4.4 will show how these tuples can be used to answer typical debugging questions.

Given a node n, all of the tuples which affected this node can be examined. The implementation of this capability will be discussed in Chapter 4, which details the KHEPERA transformation system. Next, ways of using this information to provide debugging capabilities for both the end-user and the transformation implementor will be explained.

3.4 Algorithms for User-Level Debugging

3.4.1 Overview

Assuming that the tuples outlined above are used in conjunction with a standard debugger for the target language L', they provide sufficient information to perform the following fundamental debugging functions by interactive with the tracking engine, as outlined in Figure 3.7:



$$(i, \tau, \sigma_{b_2}) \tag{3.6}$$
$$(i, \tau, \sigma_{b_2}, \text{create}, 2_2) \tag{3.7}$$
$$(i, \tau, \sigma_{b_2}, \text{copy}, (2_1), 2_2) \tag{3.8}$$
$$(i, \tau, \sigma_{b_2}, \text{replace}, \sigma_{b_2}, \sigma_{2_2}) \tag{3.9}$$
$$(i, \tau, \sigma_{b_2}, \text{delete}, b_2) \tag{3.10}$$
$$(i, \tau, \sigma_{b_2}, \text{delete}, b_1) \tag{3.11}$$
$$(i, \tau, \sigma_{b_2}, \text{delete}, =_1) \tag{3.12}$$
$$(i, \tau, \sigma_{b_2}, \text{delete}, 2_1) \tag{3.13}$$

$$(i+1, \tau', \sigma_{+_2}) \tag{3.14}$$
$$(i+1, \tau', \sigma_{+_2}, \text{create}, 3_2) \tag{3.15}$$
$$(i+1, \tau', \sigma_{+_2}, \text{replace}, \sigma_{+_2}, \sigma_{3_2}) \tag{3.16}$$
$$(i+1, \tau', \sigma_{+_2}, \text{delete}, 1_2) \tag{3.17}$$
$$(i+1, \tau', \sigma_{+_2}, \text{delete}, +_2) \tag{3.18}$$
$$(i+1, \tau', \sigma_{+_2}, \text{delete}, 2_2) \tag{3.19}$$

Figure 3.6: Example Logging of Tuples

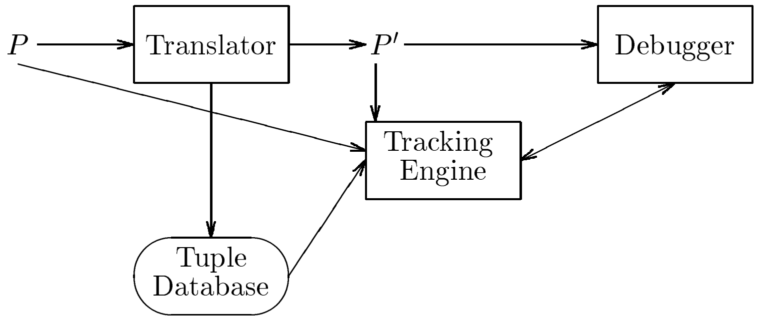


Figure 3.7: Debugging

Set a breakpoint. A syntactic element in P or a semantic element in T_k can be mapped to a line in P'.

Determine the current execution point. When a breakpoint or program exception is reached, a standard debugger will identify a line in P'. This information can be mapped backwards through the transformations, to P.

Display a value of a variable. An instance of a variable in P can be be selected and mapped to that corresponding variable or variables in P'. Transformations that performed data type changes or variable elimination can be identified and explored. This use is different from the mapping in a standard debugging system between a variable value and its location in memory. The goal here is to explain what happened to a variable during the transformation process, and to report locations in the transformed code that correspond to assignments or use of the variable in the original source code.

Display a procedure backtrace. In the same way that the current execution point is determined, each call in the procedure backtrace can be mapped to some point in T_k or P.

3.4.2 Algorithms

In this section, the algorithms necessary to answer the usual questions asked by a user-level debugger will be discussed.

When answering debugging questions about the program transformation system, a set of nodes S will be selected in some tree T_i, and a related set of nodes S' will be reported in another tree T_j.

In general, the set of nodes, S, which is being tracked should be as small as possible, growing only when a node being tracked is copied. Further, nodes being tracked should remain as closely associated as possible. Hence, the tracking algorithms will first examine node-specific tracking information, then subtree-replacement information, and will use rule subtree-matching or ancestor information as a last resort.

3.4.3 Setting a Breakpoint

When setting a breakpoint, a syntactic element in P (or a node in some T_k) must be mapped to a syntactic element in P'. The algorithm **Track-Breakpoint**, shown in Figure 3.8, describes how a set of syntactic elements S, in T_k, are tracked to other syntactic elements S', in T_ℓ. Remember that T_0 is created from the initial parse of P, and that P' is created by unparsing T_ℓ.

Usually, the user will set a breakpoint by selecting nodes in T_0. However, to have better control over the breakpoint, the user may navigate through the various intermediate trees, and set a breakpoint in some T_k.

The **Track-Node-Forward** algorithm, shown in Figure 3.9, tracks nodes between consecutive trees. This algorithm calls itself recursively since rule τ may create temporary nodes and then use copies of those nodes for insertion into the new tree. Within a single rule application, node creation (e.g., via a copy operation) is unique, so there will never be a situation where creation loops exist (e.g., node a is copied to node b and then node b is copied back to node a). Therefore, this algorithm will terminate.

Continuing the example in Figure 3.6, if the user places a breakpoint on $+_2$ in T_i, the following tracking is performed by **Track-Breakpoint**$((+_2), i, i+2)$:

Tree	Tracking	Contents of S
(start)		$+_2$
i	$+_2$ isn't changed in T_i, so S doesn't change for T_{i+1}	$+_2$
$i+1$	$+_2$ is deleted and replaced in T_{i+1}:	
	remove $+_2$ from S	$\varnothing$
	insert 3_2 into S	3_2

Therefore, in T_{i+2}, S will contain 3_2 as the node on which to set the "breakpoint". If machine code were being generated during the compilation process, this might result

Track-Breakpoint($\boldsymbol{S, k, \ell}$) returns $\boldsymbol{S'}$
Input:
 k, the index of the tree T_k where the breakpoint is specified (by the user)
 ℓ, the index of the tree on which the breakpoint should be set (by the debugger)
 S, the set of nodes in T_k on which the breakpoint should be placed
Output:
 S', the set of nodes in T_ℓ on which the breakpoint should be placed
Notes:
 $k < \ell$
 (i, τ, σ_m) describes the $T_i \to T_{i+1}$ transformation.
Algorithm:
 $t \leftarrow S$
 for i **in** $k, k+1, \ldots, \ell$ **do**
  $S' \leftarrow \varnothing$
  for n **in** t **do**
   $S' \leftarrow S' \cup$ **Track-Node-Forward($\boldsymbol{n, k}$)**
  $t \leftarrow S'$
End of Track-Breakpoint

Figure 3.8: Track-Breakpoint Algorithm

Track-Node-Forward($\boldsymbol{n, i}$) returns $\boldsymbol{A}$
Input:
    n, the node to be tracked
    i, the index of the tree T_i in which the node appears
Output:
    A, the set of nodes, in T_{i+1}, to which n tracks
Notes:
    (i, τ, σ_m) describes the $T_i \rightarrow T_{i+1}$ transformation.
    $\mathbb{S}$ is the global set of logged tuples.
    o and x are bound by the tuple match operation.
Algorithm:
    $A \leftarrow n$
    for tuples in $\mathbb{S}$ matching $(i, \tau, \sigma_m, \text{copy}, (\ldots, n, \ldots), x)$ **do**
        $A \leftarrow A \cup$ **Track-Node-Forward($\boldsymbol{x, i}$)**
    if any tuple in $\mathbb{S}$ matches $(i, \tau, \sigma_m, \text{delete}, n)$ **then**
        Remove n from A
        if any tuple in $\mathbb{S}$ matches $(i, \tau, \sigma_m, \text{replace}, \sigma_n, \sigma_x)$ **then**
                $A \leftarrow A \cup$ **Track-Node-Forward($\boldsymbol{x, i}$)**
        else if $A = \varnothing$ **then**
            if any tuple in $\mathbb{S}$ matches $(i, \tau, \sigma_m, \text{replace}, o, x) \ni n \in \sigma_o$ **then**
                $A \leftarrow A \cup$ **Track-Node-Forward($\boldsymbol{x, i}$)**
            else if $n \neq m$ **then**
                $A \leftarrow A \cup$ **Track-Node-Forward($\boldsymbol{m, i}$)**
            else
                $A \leftarrow A \cup$ **Track-Node-Forward(parent($\boldsymbol{m}$), $\boldsymbol{i}$)**

End of Track-Node-Forward

Figure 3.9: Track-Node-Forward Algorithm

in a breakpoint being set on an immediate load instruction. Alternatively, certain nodes in the output language may be denoted as nodes at which breakpoints are possible. If 3_2 wasn't this type of node, the debugger may ascend the AST in search of a more reasonable node type, perhaps placing the breakpoint on the $=_2$ node.

3.4.4 Determining the Execution Points

When determining the execution point (e.g., of an exception or a breakpoint), a syntactic element in P' (or a node in T_ℓ) must be mapped to a syntactic element in P. The algorithms **Track-Execution-Point**, shown in Figure 3.10, and **Track-Node-Backward**, shown in Figure 3.11, describe how a set of syntactic elements S, in T_ℓ, are tracked backward to another set of syntactic elements S', in T_0.

Usually, the set of syntactic elements S are generated by relating the report from the composed debugger to P', the final transformed output. Syntactic elements on P' can be related to nodes on T_ℓ. More generally, S can be selected on some arbitrary tree, T_i, and mapped back to some other arbitrary tree, T_j.

Determining an execution point is, essentially, the reverse of this forward-tracking process. Considering the example from Figure 3.6, if $=_2$ was the execution point, then **Track-Node-Backward** would have tracked this node back to the original $=_2$ in T_i. The backward tracking of 3_2 is more interesting because it must be tracked through two tree transformations:

Tree	Tracking	Contents of S
(start)		3_2
$i+1$	3_2 is created, replacing $+_2$:	
	remove 3_2 from S	$\varnothing$
	insert $+_2$ into S	$+_2$
i	$+_2$ isn't changed in T_i	$+_2$

Therefore, in T_i, S will contain $+_2$ as the node from which the execution point tracked.

3.5 Algorithms for Transformation Debugging

3.5.1 Simple Replay

The set of (k, τ, σ_m) tuples, for $k = 0, 1, 2, \ldots, \ell$, describe the complete transformation process from T_0 to T_ℓ. Iterating over these tuples and reapplying the transformations

Track-Execution-Point(S, ℓ, k) returns S'
Input:
  ℓ, the index of the tree where execution has stopped
  k, the index of the tree T_k where the execution point should be reported
  S, the set of nodes in T_ℓ on which execution has stopped
Output:
  S', the set of nodes in T_k on which the execution point should be reported
Notes:
  $k < \ell$
  $(i-1, \tau, \sigma_m)$ describes the $T_{i-1} \rightarrow T_i$ transformation.
Algorithm:
  $t \leftarrow S$
  for $i \in \ell, \ell-1, \ldots, k$ **do**
    $S' \leftarrow \varnothing$
    for$n$ **in** t **do**
      $S' \leftarrow S' \cup$ **Track-Node-Backward(n, k)**
    $t \leftarrow S'$
End of Track-Execution-Point

Figure 3.10: Track-Execution-Point Algorithm

Track-Node-Backward($\boldsymbol{n, i}$) returns $\boldsymbol{A}$

Input:

  n, the node to be tracked

  i, the index of the tree T_i in which the node appears

Output:

  A, the set of nodes, in T_{i-1}, to which n tracks

Notes:

  $(i-1, \tau, \sigma_m)$ describes the $T_{i-1} \rightarrow T_i$ transformation.

  o_1, o_2, $\ldots$, o_c, and c are bound by the tuple match operation.

Algorithm:

```
A ← n
for tuples in 𝕊 matching (i − 1, τ, σ_m, copy, (o_1, o_2, ..., o_c), n) do
    for j = 1, 2, ..., c do
        A ← A ∪ Track-Node-Backward(o_j, i)
if any tuple in 𝕊 matches (i − i, τ, σ_m, create, n) then
    Remove n from A
    if any tuple in 𝕊 matches (i − 1, τ, σ_m, replace, σ_x, n) then
            A ← A ∪ Track-Node-Backward(x, i)
    else if A = ∅ then
        if any tuple in 𝕊 matches (i − i, τ, σ_m, replace, o, x) ∋ n ∈ σ_x then
            A ← A ∪ Track-Node-Backward(o, i)
        else
            A ← A ∪ m
```

End of Track-Node-Backward

Figure 3.11: Track-Node-Backward Algorithm

provides a simple way to navigate between the trees. Since unparsing support exists for intermediate trees as well as for T_0 and T_ℓ, the intermediate views can be presented using the notations of a high-level language. The language may be an intermediate between the input language L and the output language L', but would probably be more readable than a simple Lisp-like S-expression [McCarthy 1960] rendering of the AST. The ability to provide intermediate views of the transformation process is not a feature that traditional debuggers or common transformation systems support.

Profiling data from the pre-KHEPERA PROTEUS-to-C translator indicate that the time needed to apply the transformations themselves represents less than 0.5% of the total compilation time. With the KHEPERA-based PROTEUS-to-C translator, great care has been taken so that transformation determination is rapid—hence, the translator runs faster, and transformation application requires a larger percentage of run time (roughly 40–60% in the current version).

3.5.2 Example Queries

Given the ability to replay the transformation process, and to navigate between two successive trees, the transformation implementor may want to pose debugging queries of the following form:

- The programmer may want to look at two successive ASTs and view or highlight the the updated portions.
- The programmer may want to identify "interesting" nodes on the AST and view only the transformations that involve this part.

These and other queries can be easily supported using the tracking information stored during the transformation process. Queries of this sort are not supported by any other debugging or transformation system.

3.5.2.1 Finding Updates

Given T_k and T_{k+1}, the updated nodes of the trees can be found with the **Find-Updates** algorithm shown in Figure 3.12. Considering the example from Figure 3.5 and the tree T_i, **Find-Updates(i)** returns:

$$U = (b_2, b_1, =_1, 2_1)$$
$$U' = (2_2)$$

Find-Updates($\boldsymbol{k}$) returns $(\boldsymbol{U}, \boldsymbol{U'})$

Input:

k, the index of the tree T_k which should be compared with T_{k+1}

Output:

U, the set of nodes, in T_k, which are tracked to T_{k+1}

U', the set of node, in T_{k+1}, which were tracked from T_k

Notes:

(k, τ, σ_m) describes the $T_k \rightarrow T_{k+1}$ transformation.

Algorithm:

$U \leftarrow \varnothing$

$U' \leftarrow \varnothing$

for tuples in $\mathbb{S}$ matching $(k, \tau, \sigma_m, \text{delete}, x)$ **do**

$U \leftarrow U \cup x$

for tuples in $\mathbb{S}$ matching $(k, \tau, \sigma_m, \text{copy}, (o_1, o_2, \ldots, o_c), x)$ **do**

$U \leftarrow U \cup (o_1, o_2, \ldots, o_c)$

$U' \leftarrow U' \cup x$

for tuples in $\mathbb{S}$ matching $(k, \tau, \sigma_m, \text{create}, x)$ **do**

$U' \leftarrow U' \cup x$

End of Find-Updates

Figure 3.12: Find-Updates Algorithm

3.5.2.2 Finding Next "Interesting" Transformation

Given the current tree T_k and a set of "interesting" nodes S, the **Find-Next** algorithm shown in Figure 3.13 will determine the next pair of trees, T_i and T_{i+1}, where some or all of the nodes in S are updated. For example, considering the example from Figure 3.5, **Find-Next**$(\boldsymbol{i-1}, (\boldsymbol{+_2}))$ returns $i+1$, since that is the first tree on which $+_2$ was modified.

Algorithms similar to this one allow the transformation implementor to select a set of nodes in T_k and request that the next pair of trees where some of those nodes are used or changed be displayed. This algorithm finds the pair of trees where nodes in S were deleted or copied. Other algorithms which are useful for debugging might be ones that find only the trees where deletion took place, or only trees where *all* of the nodes in S were used or destroyed.

Find-Next($\boldsymbol{k, S}$) returns $\boldsymbol{i}$
Input:
  k, the index of the tree T_k that should be compared with T_{k+1}
  S, the set of nodes in T_k that should be tracked until copied or destroyed
Output:
  i, the index of the tree T_i where nodes in S were copied or destroyed
Notes:
  (k, τ, σ_m) describes the $T_k \rightarrow T_{k+1}$ transformation.
  If, at algorithm termination, $i = \ell$, then none of the nodes in S were used or deleted between T_k and T_ℓ.
Algorithm:
  for $i \in k, k+1, \ldots, \ell$ **do**
    if any tuple in $\mathbb{S}$ matches $(i, \tau, \sigma_m, \text{delete}, n) \ni n \in S$ **return** i
    if any tuple in $\mathbb{S}$ matches $(i, \tau, \sigma_m, \text{copy}, (\ldots, n, \ldots), x) \ni n \in S$ **return** i
End of Find-Next

Figure 3.13: Find-Next Algorithm

3.5.2.3 Other Queries

The **Track-Node-Forward** and **Track-Node-Backward** algorithms can be also be used to debug the transformation system. These algorithms can answer questions about how a node was transformed between the input program and the output program, or how a syntactic element in the final transformed version of the program relates to the original input program.

3.6 Conclusion

In this chapter, algorithms for tracking information and answering debugging queries were presented. These algorithms are designed for use in a high-level language processor built using a transformational programming system based on tree manipulation. Tracking of debugging information is performed at the lowest levels of tree manipulation in a manner that is independent of the semantics of the transformations being applied.

The information tracked can be used to answer questions from a composed debugger to allow breakpoint setting or execution point determination. Questions from the debugger about variable values can be answered, thereby helping the composed debugger to explain program transformations which might change the expected value of a variable.

Further, the information can be used to answer questions that will help debug the transformation system itself. These queries can be used by the transformation implementor during the implementation and debugging phase of the program translator, or they can help a sophisticated end-user understand how the transformation system changed the program being debugged.

The next two chapters will:

- Outline KHEPERA, a prototype implementation of a transformation system that performs the tracking outlined here, including slightly more complicated transformational examples.

- Explore substantially more complicated examples taken from our work with PROTEUS.

Chapter 4

The KHEPERA Transformation System

The KHEPERA system is a toolkit for the rapid implementation and long-term maintenance of research compilers and processors for domain-specific languages (DSLs) [Faith et al. 1997]. KHEPERA emphasizes the construction of processors which translate from one high-level language to another. So, while KHEPERA may be useful for implementing front-ends for more general compilers, it does not currently provide specialized support for code generation. KHEPERA supports the viewpoint that program translators are most easily implemented with simple parsing, sophisticated tree-based analysis and manipulation, and target source generation using pretty-printing techniques.

In the context of this dissertation, the KHEPERA system fills two main roles.

- First, the system provides an implementation of the algorithms outlined in Chapter 3: the source-to-source transformation support provided by KHEPERA transparently tracks debugging information, providing support for transformation replay and navigation, and for answering debugger queries.
- Second, KHEPERA provides support and scaffolding necessary for experimentation with performance-optimized tree-traversal and transformation algorithms. At the end of this chapter, I present an algorithm for rapid tree traversal.

4.1 Goals for a Program Transformation Toolkit

The implementation of a program translator requires considerable overhead, both for the initial implementation and as the language specification evolves. A toolkit for the construction of translators should leverage existing, familiar tools as much as possible. Use of such tools takes advantage of previous implementor knowledge and the availability of comprehensive resources explaining these tools.

Within a transformational model, a translator-building toolkit can simplify the implementation process by providing specialized tools where pre-existing tools are not already available, and by providing integrated support for debugging within this framework.

The KHEPERA system facilitates both the problem of rapid translator prototyping and the problem of long-term translator maintenance through the following specific design goals:

Familiar, modularized parsing components. KHEPERA supports the use of familiar scanning and parsing tools (e.g., the traditional `lex` and `yacc` [Levine et al. 1992], or the newer PCCTS [Parr 1997]) for implementation of a language processor. Because KHEPERA concentrates on providing the "missing pieces" that help with rapid implementation of language processors, previous programmer knowledge can be utilized, thereby decreasing the slope of the necessary learning curve.

Familiar, flexible, and efficient semantic analysis. KHEPERA uses the source-to-source transformational model outlined in Figure 4.1. This model uses tree-pattern matching for AST manipulation, analysis, and attribute calculation. For tedious but common tasks, such as tree-pattern matching, sub-tree creation, and sub-tree replacement, KHEPERA provides a "little language" [Bentley et al. 1987; Bentley 1988] for describing tree matches and for building trees. For unpredictable or language-specific tasks, such as attribute manipulation or analysis, the KHEPERA little language provides an escape to a familiar general-purpose programming language (C). Standard tree traversal orders are supported (e.g., preorder, postorder), as well as arbitrarily complicated syntax-directed sequencing. Data-structure maintenance accelerates pattern matches in standard tree traversal orders.

Familiar output mechanism. A pretty-printing facility is provided that can output the AST in an easily readable format at any time. One strong advantage of this pretty-printer when compared with other systems is that it will always be able to print the AST, regardless of how much of the transformation has been performed. If the AST is in the original input format or the final output format, then the pretty-printed program will probably be executable in the input language L or the output

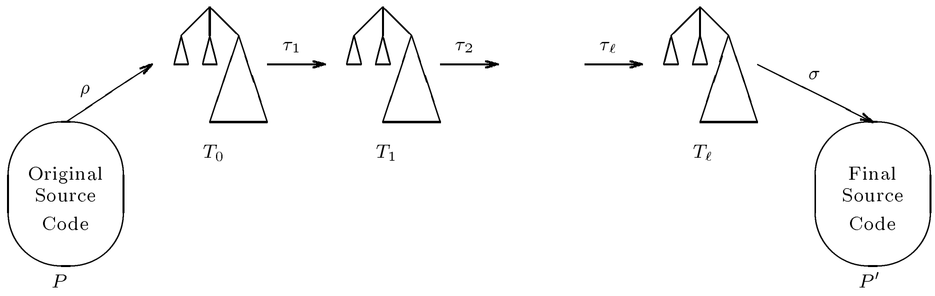


Figure 4.1: Transformation Process

language L'. However, if the AST being printed is one of the intermediate trees, then the output will use some combination of the syntax of L and L', with a fallback to simple Lisp-like S-expressions [McCarthy 1960] for AST constructs which do not have well-defined concrete syntax. While the program printed may not be executable, it does use a familiar syntax that is helpful for a human who is familiar with both languages when replaying transformations during the debugging process.

Debugging support for language translation. KHEPERA implements the tracking algorithms described in Section 3.3.2, and includes a viewer which uses the debugging algorithms from Section 3.4 to replay the transformation sequence and answer questions about which transformations were applied at which points on the AST. This is helpful when writing and debugging the language processor, as well as when implementing a debugger for programs written in the experimental language itself.

Transformations are either written in the high-level KHEPERA language and are transformed by KHEPERA into executable C with calls to the KHEPERA library (as discussed in Section 4.4.7 and shown in Figure 4.9 and Figure 4.10); or the transformations are written using explicit calls to the KHEPERA library tree manipulation functions. In either case, low-level hooks in the KHEPERA library track debugging information when nodes or subtrees are created, destroyed, copied, or replaced. This low-level information can be analyzed using the algorithms from Chapter 3 to provide the ability to navigate through intermediate versions of the transformed program, and the ability to answer specific queries that support the debugging of the final transformed output:

- setting breakpoints,

- determining current execution location (e.g., in response to a breakpoint or program exception), and
- tracking variable use and transformation.

The tracking algorithms were presented in Chapter 3, a short example of how the tracking data can be used to set a breakpoint will be shown in Section 4.4, and more extensive debugging problems will be explored in Chapter 5.

4.2 Related Work

KHEPERA is similar to some compiler construction kits. However, these systems usually restrict the scanning and parsing tools used [Grosch and Emmelmann 1990; Bates 1996]; specify AST transformations using a low-level language, such as C [Tjiang et al. 1992] (instead of a high-level transformation-oriented language); or require that the AST always conforms to a single grammar specification, making translation from one language to another difficult [Reasoning Systems 1990]. Several systems share several of the limitations listed above, often because they provide some interesting feature that is peripheral to the task of source-to-source transformation.

For example, TXL [Cordy et al. 1991; Cordy and Carmichael 1993], while not a complete compiler generation tool, is designed to perform source-to-source transformations to provide a means of rapid prototyping of language extensions. The language used to specify these transformations uses concrete syntax and depends on the use of the TXL parser, a top-down, fully backtracking parser that can handle any context-free grammar. Using the input grammar for an "unparser" provides a means of printing the output program. However, this dependence on a single input grammar restricts the use of TXL to same-language transformations. Also, the grammar can be difficult to write, since a poorly-constructed grammar can cause the parse to take a long time to complete.

SORCERER, from the PCCTS toolkit [Parr 1997], is the most similar to KHEPERA, since it does not require the use of specific scanning and parsing tools, and since it provides a little language in the style of `lex` and `yacc` with embedded procedures written in another general-purpose programming language (e.g., C). SORCERER and KHEPERA share abilities to describe tree structures and perform syntax-directed translations; both support the writing of AST-based interpreters. In contrast, KHEPERA also supports rule-based translations that do not require a grammar specifi-

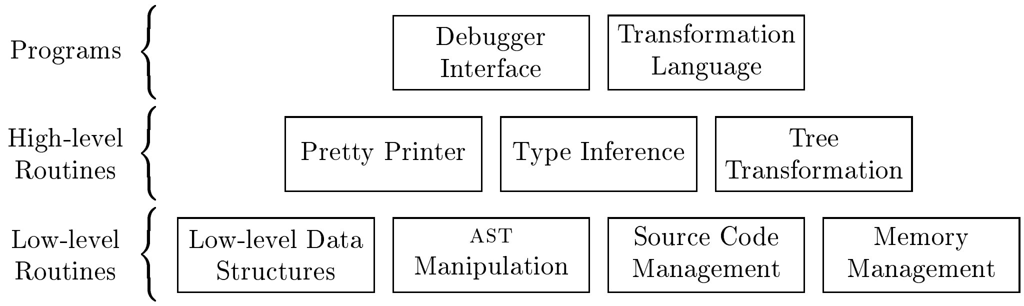


Figure 4.2: The KHEPERA Transformation System

cation for the AST; KHEPERA rules are suitable for writing compiler-required analysis routines; and writing pretty-printer rules in KHEPERA does not require a complete tree-grammar specification. This allows pretty-printing to easily take place during grammar evolution.

None of the previous systems, including SORCERER, contain built-in support for "replay" of transformations, or for automatic and transparent tracking of debugging information. The transformation discovery and replay capabilities of KHEPERA have been used to implement a viewer that presents intermediate views of the transformation process, and that can answer typical queries posed by a debugger (see Section 4.4.8).

4.3 Overview of KHEPERA

The KHEPERA library provides low-level support for:

- building an AST
- applying transformation rules to the AST
- unparsing the P' source code from the T_ℓ AST (the σ "transformation")

An overview of the KHEPERA system is shown in Figure 4.2. KHEPERA encapsulates low-level details of the language processor implementation: AST manipulation, symbol and type table management, and management of line-number and lexical information [Faith 1996a]. On a higher level, library routines are available to support pretty-printing (currently, with a small language to describe how to print each node type in the AST), type inference, and tree transformation. The tree transformation routines include functions for tracking debugging information, as described in Chapter 3 [Faith 1996b]. Further, the implementation of a little language (described in Appendix A) supports a high-level description of the transformation rules. If transformation rules are written in the KHEPERA language, or if they are written in an ad hoc manner using the underlying KHEPERA AST manipulation library, then the debugging tracking and transformation replay support will be automatically provided.

An overview of how the KHEPERA system fits into a complete language translator implementation solution is shown in Figure 4.3. In the example shown in the next section, various input specifications will be outlined. In general, various specifications are written that are processed by various intermediate processors. Some of these processors may be familiar tools, such as `lex` and `yacc`. Others are new tools contained in the KHEPERA toolkit. These processors generate, in this case, C code that is then compiled by a native compiler, producing a language processor for the specified language.

The input specifications to the traditional processors, such as `lex` and `yacc`, make calls to the KHEPERA library routines to track source line number and token offset information, and to build the initial AST. The input specifications to the KHEPERA processors describe how to manipulate and print the AST. Some of these specifications are optional. For example, if the `lex` specification doesn't make all of the necessary calls to the KHEPERA library, it may be impossible for KHEPERA to provide line number information later in the transformation process. However, without this information, other features of KHEPERA will still be provided. For the initial language implementation, the programmer may find it convenient to leave out a complete type inference or pretty-printing specification, relying on default behavior or assumptions about the experimental language (e.g., that all of the variables are integers and do not need type checking). As the experimental language becomes more complicated, or as the implementation becomes more complete, these other specifications can be added or enhanced as needed.

In Figure 4.4, the "Language Processor" from Figure 4.3 is expanded, showing the components that are created from the language processor source code and showing how the language processor executable is used during the compilation of a program written in the experimental language. The input program is parsed, transformed, and pretty-printed for compilation with a native compiler. All of the components of the language processor make calls to the KHEPERA library, which provides support for high-level functions, such as tree-manipulation and pretty-printing, as well as

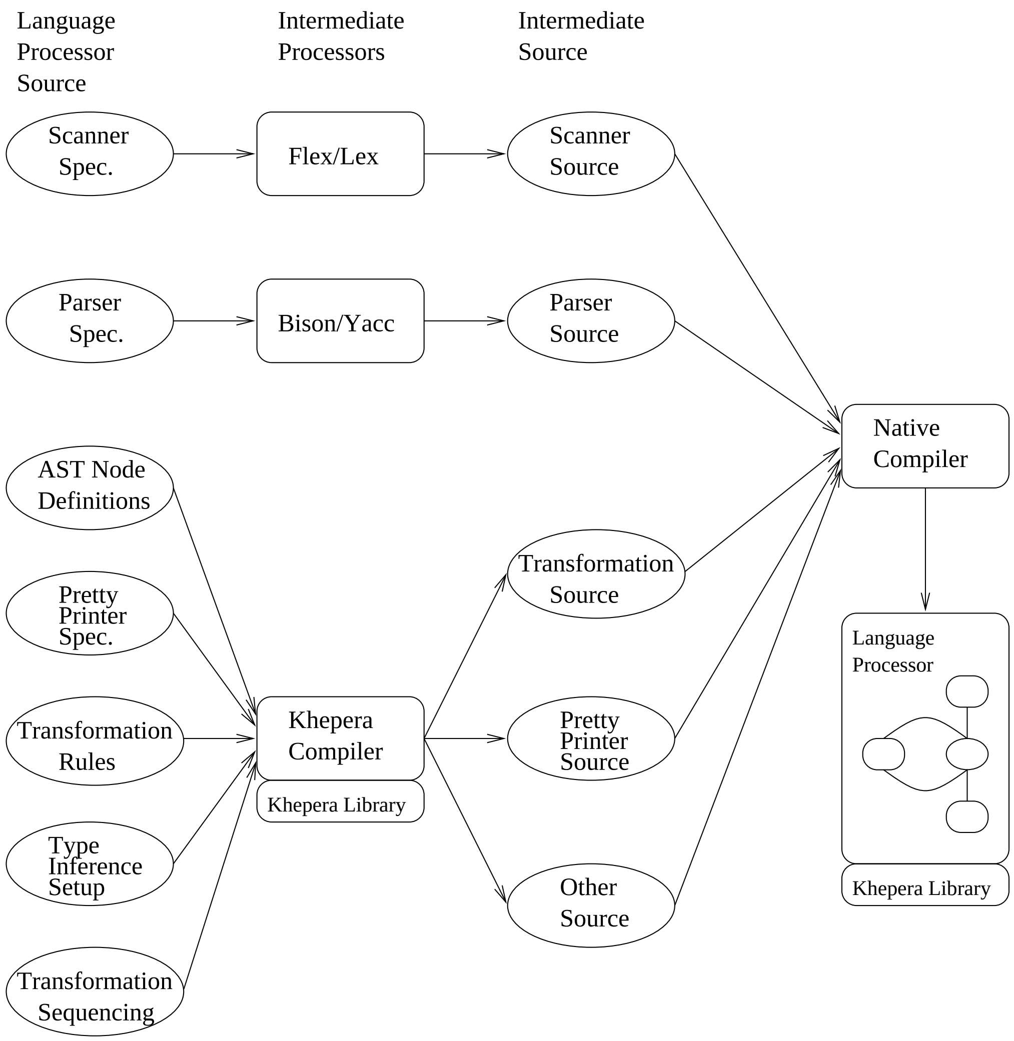


Figure 4.3: Using the KHEPERA Transformation System

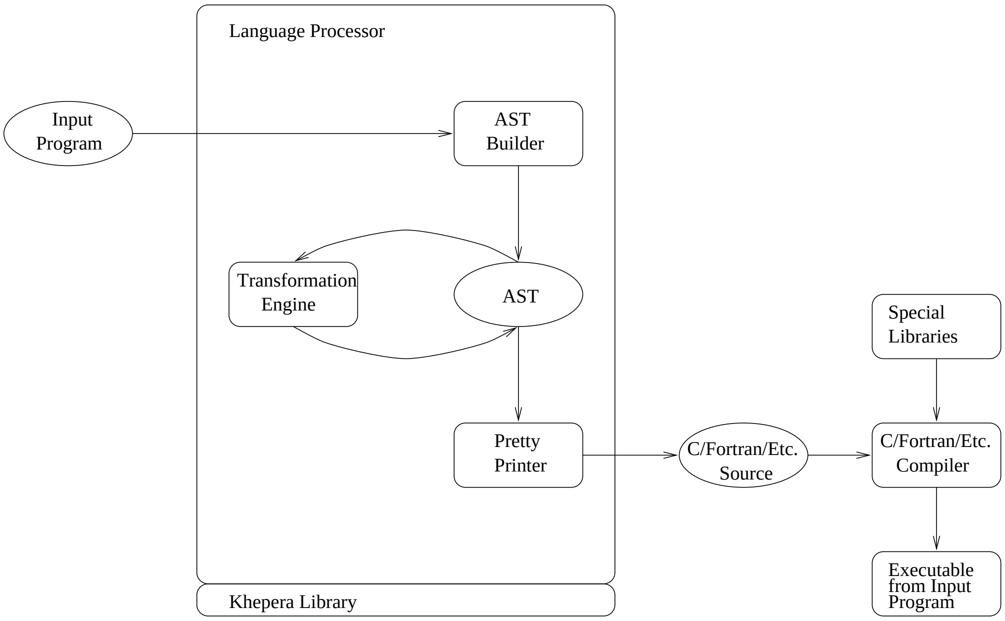


Figure 4.4: Using the Language Processor

extensive support for low-level functions required by compiler implementors, such as string pool or symbol table maintenance.

4.4 Example

A simple language translation problem based on Proteus [Prins and Palmer 1993; Riely et al. 1995] will be used to illustrate the Khepera system. This example language is a subset of Fortran 90 [Adams et al. 1992] with the addition of a nested sequence data type and a *sequence comprehension* construct that can be used to create nested sequences. The translation problem is to remove all sequence comprehension constructs and replace them with simple data-parallel operations, yielding a program suitable for compilation with a standard-conforming Fortran 90 compiler.

4.4.1 Example Language Syntax

The lexical elements of the experimental language are:

Id **Int** `(/ /) ( ) + , : = in`

$$
\begin{array}{rcl}
\textit{program} & ::= & \textit{statement-list} \\
\textit{statement} & ::= & \textbf{Id} \texttt{ = } \textit{expression} \\
\textit{statement-list} & ::= & \textit{statement} \\
 & | & \textit{statement-list}\ \textit{statement} \\
\textit{expr} & ::= & \textbf{Id} \\
 & | & \textbf{Int} \\
 & | & \textit{expr}\ \texttt{+}\ \textit{expr} \\
 & | & \texttt{add(}\ \textit{depth}\ \texttt{,}\ \textit{expr}\ \texttt{)} \\
 & | & \texttt{length(}\ \textit{depth}\ \texttt{,}\ \textit{expr}\ \texttt{)} \\
 & | & \texttt{range(}\ \textit{depth}\ \texttt{,}\ \textit{expr}\ \texttt{)} \\
 & | & \texttt{dist(}\ \textit{depth}\ \texttt{,}\ \textit{expr}\ \texttt{,}\ \textit{expr}\ \texttt{)} \\
 & | & \texttt{(/}\ \textit{expr-list}\ \texttt{/)} \\
 & | & \texttt{(/}\ \textbf{Id}\ \texttt{in}\ \textit{expr}\ \texttt{:}\ \textit{expr}\ \texttt{/)} \\
\textit{expr-list} & ::= & \textit{expr} \\
 & | & \textit{expr-list}\ \textit{expr} \\
\textit{depth} & ::= & \texttt{depth=}\textbf{Int}
\end{array}
$$

Figure 4.5: CFG for First Example Language

A program is described by the context free grammar (CFG) shown in Figure 4.5. For this example, the array constructor notation from FORTRAN 90 is used to specify literal sequences and a similar notation is used to specify the sequence comprehension construct. However, the sequence comprehension construct creates arbitrarily nested, *irregular* sequences. (In contrast, the array constructor from FORTRAN 90 can only generate vectors or rectangular arrays.)

4.4.2 Example Language Semantics

As a convenience, every value in the example language is considered an element of a sequence type. A sequence type includes a scalar base type and a depth. For simplicity, only integer scalar types are considered. Zero-depth sequences are simply scalars. Non-scalar sequences (i.e., with depth ≥ 1) are written as lists of elements between `(/` and `/)` brackets; for example, `(/ /)` is the empty sequence, and `(/ (/` $1,2$ `/), (/` $3,4,5$ `/), (/ /) /)` is a sequence of three elements, a "sequence of sequences of integers". All sequences have uniform depth.

Omitted here is a collection of type (inference) rules for the language that define a well-typed program (these rules would be trivial for this example, since only integer scalar types are permitted). See Hindley [1969], Milner [1978], and Cardelli [1987] for detailed information on polymorphic type systems.

4.4.2.1 Primitive Operations

Primitive operations of arity ℓ are applied by writing $p($`depth=` $d, a_1, \ldots, a_\ell)$, where p is a primitive operation (`add`, `length`, `range`, or `dist`), $a_1, \ldots, a_\ell$ are the arguments, and d is the depth at which the operation is to be applied. If d is zero, the application is *basic*, otherwise it is *lifted* [Riely et al. 1995]. To avoid error, the nesting structures of the arguments must be identical down to depth d. For example,

$$
\begin{aligned}
\texttt{add(depth=}\ 0, 5, 6\ \texttt{)} &= 11 \\
\texttt{add(depth=}\ 1, \texttt{(/}\ 4,3,1\ \texttt{/)}, \texttt{(/}\ 3,6,7\ \texttt{/)}\ \texttt{)} &= \texttt{(/}\ 7,9,8\ \texttt{/)} \\
\texttt{add(depth=}\ 2, \texttt{(/ (/ /)}, \texttt{(/}\ 2,3\ \texttt{/) /)}, & \\
\texttt{(/ (/ /)}, \texttt{(/}\ 7,1\ \texttt{/) /))} &= \texttt{(/ (/ /)}, \texttt{(/}\ 9,4\ \texttt{/) /)}
\end{aligned}
$$

Below, I give extensional descriptions of the sequence primitives:

- `add` performs *addition* on the elements of a sequence, returning a sequence with the same depth as the two arguments. A special notation with `depth=` 0, using infix notation is allowed: $a + b \stackrel{def}{=}$ `add`$($`depth=` $0, a, b)$. Examples are shown above.

- `length` returns the *length* of its argument. For example:

$$
\begin{aligned}
\texttt{length(depth=}\ 0, \texttt{(/}\ 9,8,7,6\ \texttt{/))} &= 4 \\
\texttt{length(depth=}\ 0, \texttt{(/ (/}\ 9,8,7\ \texttt{/)}, \texttt{(/}\ 6,5\ \texttt{/) /))} &= 2 \\
\texttt{length(depth=}\ 1, \texttt{(/ (/}\ 9,8,7\ \texttt{/)}, \texttt{(/}\ 6,5\ \texttt{/) /))} &= \texttt{(/}\ 3,2\ \texttt{/)}
\end{aligned}
$$

- `range` is the *iota* function from APL. For any value of n and all integer values of d,

$$\texttt{length(depth=}\ d, \texttt{range(depth=}\ d, n\ \texttt{))} = n$$

  For example:

$$
\begin{aligned}
\texttt{range(depth=}\ 0, 5\ \texttt{)} &= \texttt{(/}\ 1,2,3,4,5\ \texttt{/)} \\
\texttt{range(depth=}\ 1, \texttt{(/}\ 2,3\ \texttt{/))} &= \texttt{(/ (/}\ 1,2\ \texttt{/)}, \texttt{(/}\ 1,2,3\ \texttt{/) /)}
\end{aligned}
$$

- `dist` *distributes* a value, making a number of copies. For all values of c and n, and for all integer values of d,

$$\texttt{length(depth=}\ 0, \texttt{dist(depth=}\ d, c, n\ \texttt{)}\ \texttt{)} = n$$

  For example:

$$\begin{aligned} \texttt{dist(depth=}\ 0, 1, 5\ \texttt{)} &= \texttt{(/}\ 1, 1, 1, 1, 1\ \texttt{/)} \\ \texttt{dist(depth=}\ 1, \texttt{(/}\ 1, 2\ \texttt{/)}, \texttt{(/}\ 2, 3\ \texttt{/)}\ \texttt{)} &= \texttt{(/}\ \texttt{(/}\ 1, 1\ \texttt{/)}, \texttt{(/}\ 2, 2, 2\ \texttt{/)}\ \texttt{/)} \end{aligned}$$

4.4.2.2 Sequence Comprehension

For an expression, e with free variable i, the sequence comprehension

$$\texttt{(/}\ i\ \texttt{in}\ A\ :\ e(i)\ \texttt{/)}$$

yields the sequence of successive values of e obtained when i is bound to successive values in A. For example, the sample program:

```
A = range(depth = 0, 3);
B = (/ i in A: i + i /);
C = (/ i in A: (/ j in range(depth = 0, i): i /) /)
```

yields:

```
A = (/ 1, 2, 3 /)
B = (/ 2, 4, 6 /)
C = (/ (/ 1 /),
       (/ 2, 2 /),
       (/ 3, 3, 3 /) /)
```

4.4.3 Example Translation

A program is viewed in terms of the AST corresponding to the CFG of Section 4.4.1. In the AST, an application of one of the four basic operations is written as a function application node (`N_Call`) with the operation to be applied described by the left-most child and a *depth* attribute that is 0. The other children of the node are expressions for each of the arguments.

The following 3 rules can be used to eliminate all sequence comprehension constructs from the AST:

Rule 1

$$\texttt{(/}\ x_1\ \texttt{in}\ e_1\ \texttt{:}\ x_1\ \texttt{/)} \longrightarrow e_1$$

Rule 2 Provided e_2 is an **Id** or **Int**, and $e_2 \neq x_1$,

$$\texttt{(/}\ x_1\ \texttt{in}\ e_1\ \texttt{:}\ e_2\ \texttt{/)} \quad\longrightarrow\quad \texttt{dist(depth=}\ 0, e_2, \texttt{length(depth=}\ 0, e_1\texttt{))}$$

Rule 3 Provided p is a primitive operation (`add`, `length`, `range`, or `dist`),

$$\begin{array}{lcl} \texttt{(/}\ x_1\ \texttt{in}\ e_0\ \texttt{:} & & p\texttt{(depth=}\ d+1, \\ \qquad p\texttt{(depth=}\ d, & \longrightarrow & \quad \texttt{(/}\ x_1\ \texttt{in}\ e_0\ \texttt{:}\ e_1\ \texttt{/)}, \\ \qquad\quad e_1, \ldots, e_n\ \texttt{) /)} & & \quad \ldots, \\ & & \quad \texttt{(/}\ x_1\ \texttt{in}\ e_0\ \texttt{:}\ e_n\ \texttt{/))} \end{array}$$

The resultant AST can be written out as FORTRAN 90. Given an appropriate implementation of the primitive functions, the resultant program specifies fully parallel execution of each sequence comprehension construct, regardless of the degree of nesting and sequence sizes.

For example, using these rules, the program from Section 4.4.2.2 (page 71) would be transformed as follows:

```
A = range(depth=0, 3)
B = add(depth=1, A, A)
C = dist(depth=1, A, length(depth=1, range(depth=1, A)))
```

Note that functions with $\texttt{depth} = 0$ operate on scalar arguments, whereas functions with $\texttt{depth} \geq 1$ operate on nested sequence arguments.

When the source language is more expressive and optimization becomes an issue, the rules shown here are not necessarily terminating, hence additional sequencing rules must be added to control rule application [Palmer 1996].

```
NL              \n
...
%%
<STARTOFLINE>{
   .*{NL}       src_line(yytext,yyleng); yyless(0); BEGIN(OTHER);
   .*           src_line(yytext,yyleng); yyless(0); BEGIN(OTHER);
}
...
{NL}            BEGIN(STARTOFLINE);
```

Figure 4.6: Storing Lines While Scanning

4.4.4 Scanner

The AST is constructed using a scanner and parser generator of the implementor's choice with calls to the KHEPERA library AST construction routines. At the level of the scanner, KHEPERA provides support for source code line number and token offset tracking. This support is optional, but is very helpful for debugging. If the implementor desires line number and token offset tracking, the scanner must interact with KHEPERA in several ways.

First, each line of source code must be registered. In versions of `lex` that support states, providing this information is trivial (although inefficient), as shown in Figure 4.6: each line is captured in the `STARTOFLINE` state, and then the lexer's input buffer is reset so that the `OTHER` state can parse the tokens in the line. For other scanner generators, or if scanning efficiency is of great concern, other techniques can be used. The routine `src_line` stores a copy of the line using low-level string-handling support. While the routines used in these examples are tailored for `lex` semantics, the routines are generally wrapper routines for lower-level KHEPERA functions and would, therefore, be easy to implement for other front-end tools.

Second, line number information generated by the C preprocessor must be interpreted correctly. This requires a simple `lex` action:

```
^#\ .*      src_cpp_line(yytext, yyleng);
```

Finally, every scanner action must advance a pointer to the current position on the current line. This is accomplished by having every action make a call to `src_get(yyleng)`, a minor inconvenience that can be encapsulated in a macro.

```
%token <token_from_scanner> '='
%type  <ast_node> Statement StatementList
%type  <ast_node> Identifier Expression
Statement: Identifier '=' Expression
           { $$ = tre_mk(N_Assign, $2.src, $1, $3, 0); };

StatementList: Statement
                 { $$ = tre_mk(N_StatementList,
                              tre_src($1), $1, 0); }
               | StatementList Statement
                 { $$ = tre_append($1, $2); };
```

Figure 4.7: Building the AST While Parsing

4.4.5 Parsing and AST Construction

The productions in the parser call KHEPERA tree-building routines—all other work can be reserved for later tree traversal. This tends to simplify the parser description file, and allows the implementor to concentrate on parsing issues during this phase of development. A few example `yacc` productions are shown in Figure 4.7. The second argument to `tre_mk` is a pointer to the (optional) source position information obtained during scanning. The abstract representation of the constructed AST is that of an n-ary tree, and routines are available to walk the tree using this viewpoint (physically, the tree is stored as a tilted binary tree, although other underlying representations would also be possible).

Immediately after the parsing phase, the AST is available for printing. Without any pretty-printer description, the AST is printed as a nested S-expression, as shown in Figure 4.8.

4.4.6 Pretty-Printing

For pretty-printing, KHEPERA uses a modification of the algorithm presented by Oppen [1980]—the main modification allows the algorithm to continue formatting if the specified line length is exceeded and a break cannot be found. This algorithm is linear in space and time, and does not backtrack when printing. The implementation was straightforward, with simple modifications added to support source line tracking and formatted pretty-printing. Other algorithms for pretty printing, some of which

```
A = range(depth=0, 3)
B = (/ i in A : i + i /)
C = (/ i in A :
        (/ j in range(depth=0, i) : i /) /)
```

(a) Original Program

```
(N_StatementList
  (N_Assign
    (N_Identifier/"A")
    (N_Call
      (N_Identifier/"range")
      (N_ExpressionList
        (N_Integer/3))))
  (N_Assign
    (N_Identifier/"B")
    (N_SequenceBuilder
      (N_Iterator
        (N_Identifier/"i")
        (N_Identifier/"A"))
      (N_Add
        (N_Identifier/"i")
        (N_Identifier/"i"))))
  (N_Assign
    (N_Identifier/"C")
    (N_SequenceBuilder
      (N_Iterator
        (N_Identifier/"i")
        (N_Identifier/"A"))
      (N_SequenceBuilder
        (N_Iterator
          (N_Identifier/"j")
          (N_Call
            (N_Identifier/"range")
            (N_ExpressionList
              (N_Identifier/"i"))))
        (N_Identifier/"i")))))
```

(b) Initial AST (with attribute values shown after the slash)

Figure 4.8: Example Input and Initial AST

support a finer-grain control over the formatting, are presented by Rubin [1983], Pugh and Sinofsky [1987], Cameron [1988], Jokinen [1989], and Ruckert [1997].

For each node type in the AST, a short description, using `printf`-like syntax, tells how to print that node and its children. If the node can have several different numbers of children, several descriptions may be present, one for each variation. List nodes may have an unknown number of children. Multiple descriptions may be present for multiple languages, with "fallback" from one language to another specified at printing time (so, Fortran may be printed for all of those nodes that have Fortran-specific descriptions, with initial fallback to C, and with final fallback to generic S-expressions). This fallback scheme provides usable pretty-printing during development, even before the complete pretty-printer description is finished and debugged.

For printing which requires local analysis, implementor-defined functions can be used to return pre-formatted information or to force a line break. These functions are passed a pointer to the current node, so they have access to the complete AST from the locus being printed. While the pretty-printer is source-language independent and is unaware of the specific application-defined attributes present on the AST, the implementor-defined functions have access to all of this information. These functions are typically used to format type information or to add comments to the generated source codes.

Additional pretty-printer description syntax allows line breaks to be declared as "inconsistent" or "consistent"[1]; allows for forced line breaks; and permits indentation adjustment after breaks.

---

[1]See Oppen [1980] for details. Each group may have several places where a break is possible. An inconsistent break will select one of those possible places to break the line, whereas a consistent break will select *all* of these places if a break is needed anywhere in the group. This allows the following formatting to be realized (assuming breaks are possible before +):

Inconsistent	Consistent
x = a + b + c + d + e + f	x = a + b + c + d + e + f

4.4.7 Using the KHEPERA Transformation Language

KHEPERA transformations are specified in a special little language that is translated into C code for tree-pattern matching and replacement. A simple transformation rule conditionally matches a tree, builds a new tree, and performs a replacement. This language is described more fully in Appendix A. In this section, a subset of this language will be discussed in the context of the the example currently being developed.

The KHEPERA rule that implements the first sequence comprehension elimination transformation (Rule 1 from Section 4.4.3) is shown in Figure 4.9, together with a formal description of the rule, using the variable names from the rule implementation, and an example AST.

In Figure 4.9c, a tree pattern follows the **match** keyword. Tree patterns are written as S-expressions. The tree pattern in this example is compiled to the pattern matching code shown in the first part of Figure 4.10 (code for sections of the rule follow the comment containing that section).

The **when** expression, which contains arbitrary C code, guards the match, preventing the rest of the rule from being executed unless the expression evaluates to true. The **build** statement creates a new subtree, taking care to copy subtrees from the matched tree, since those subtrees are likely to be deleted by a **replace** command.

The tracking necessary for debugging and transformation replay is performed at a low level in the KHEPERA library. However, the KHEPERA language translator automatically adds functions (with names starting with `trk_`, shown with boxes in Figure 4.10) to the generated rules:

`trk_enter`, `trk_leave` These functions ensure that, when a rule makes nested calls to other rules, all of the associated changes to the AST are "charged" to the outermost rule in the sequence. This is essential for replay, using the logging described by Tuple 3.1 (page 47).

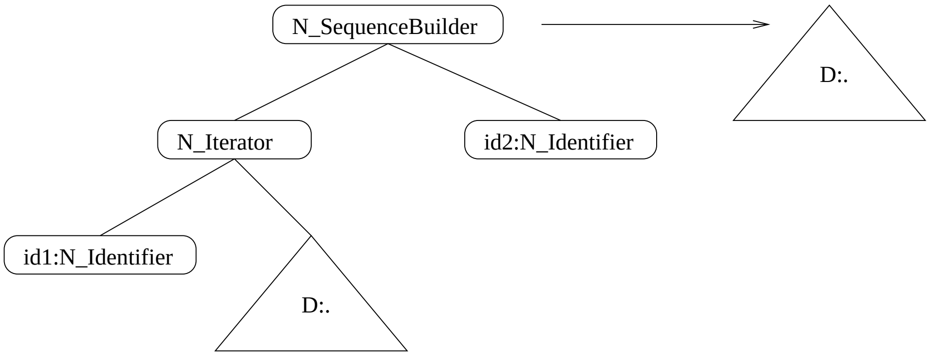
`trk_application` This rule registers the name of the rule currently being applied (τ) and the subtree matched by the rule (σ_m). This information is used by the low-level AST-manipulation functions of the KHEPERA library to log the tuples described in Tuple 3.2, Tuple 3.3, Tuple 3.4, and Tuple 3.5 (page 47).

`trk_work` This rule causes the tuple described in Tuple 3.1 (page 47) to be logged when a KHEPERA **replace**, **do**, or **delete** command actually causes a modifi-

If $\mathrm{id}_1 = \mathrm{id}_2$, then

$$\overbrace{\texttt{(/}\ \underbrace{\mathrm{id}_1\ \texttt{in}\ D}_{\text{N\_Iterator}}\ :\mathrm{id}_2\ \texttt{/)}}^{\text{N\_SequenceBuilder}} \longrightarrow D$$

(a) Formal Rule 1



(b) AST Transformation

```
rule eliminate_iterator1
{
   match (this:N_SequenceBuilder
           (N_Iterator id1:N_Identifier D:.)
           id2:N_Identifier)
   when (tre_symbol(id1) == tre_symbol(id2))
   build new with D
   replace this with new
}
```

(c) Khepera Rule

Figure 4.9: Simple Transformation Rule (Rule 1)

```
int rule_eliminate_iterator1( int *_kh_flag, tre_Node _kh_node )
{
   const char *_kh_rule = "rule_eliminate_iterator1";
   Node _kh_pt;
   Node this = NULL; /* sym */
   Node id1 = NULL; /* sym */
   Node D = NULL; /* sym */
   Node id2 = NULL; /* sym */
   Node new = NULL;

   /* match (this:N_SequenceBuilder
             (N_Iterator id1:N_Identifier D:.) id2:N_Identifier) */
   trk_enter();
   _kh_pt = _kh_node;
   if (_kh_pt && tre_id( this = _kh_pt ) == N_SequenceBuilder) {
      _kh_pt = tre_child( _kh_pt ); /* N_Node */
      if (_kh_pt && tre_id( _kh_pt ) == N_Iterator) {
         _kh_pt = tre_child( _kh_pt ); /* N_Node */
         if (_kh_pt && tre_id( id1 = _kh_pt ) == N_Identifier) {
            _kh_pt = tre_right( _kh_pt );
            if (_kh_pt) {
               D = _kh_pt;
               _kh_pt = tre_parent( _kh_pt );
               _kh_pt = tre_right( _kh_pt );
               if (_kh_pt && tre_id( id2 = _kh_pt ) == N_Identifier) {
                  _kh_pt = tre_parent( _kh_pt );
                  assert( _kh_pt == _kh_node );

                  /* when (tre_symbol(id1) == tre_symbol(id2)) */
                  if (tre_string(id1) == tre_string(id2)) {
                     trk_application( _kh_rule, _kh_node );

                     /* build new with D */
                     new = tre_copy(D);

                     /* replace this with new */
                     ++*_kh_flag;
                     trk_work( _kh_rule, _kh_node );
                     tre_replace( this, new );
                  }
               }
            }
         }
      }
   }
   trk_exit();
   return 0;
}
```

Figure 4.10: Generated Tree-Pattern Matching Code (Rule 1)

cation to the AST. (The arguments are the same as for `trk_application`—this redundancy is used for error detection.)

For completeness, the second rule, Rule 2 from Section 4.4.3, is shown in Figure 4.11.

Finally, a more complicated KHEPERA rule is shown in Figure 4.12, with the corresponding AST transformation shown in Figure 4.13. This rule implements the third sequence comprehension elimination transformation (Rule 3 from Section 4.4.3). This transformation matches a `N_SequenceBuilder` with a function call, then iterates over the arguments to the function call, building up the new arguments for the new call to the promoted function. After the new function call is created, the *depth* attribute (called `prime` in the KHEPERA code) is updated.

The example in Figure 4.12 uses the **children** statement to iterate over the children of the `N_ExpressionList` node, and uses the **do** statement as a general-purpose escape to C. This escape mechanism is used to build up a new list with the `tre_append` function, and to modify an implementor-defined attribute (`prime`).

KHEPERA language features not discussed here include the use of a conditional **if-then-else** statement in place of a **when** statement, the ability to break out of a **children** loop, and the ability to perform tree traversals of matched subtree sections (this is useful when an expression must be examined to determine if it is independent of some variable under consideration).

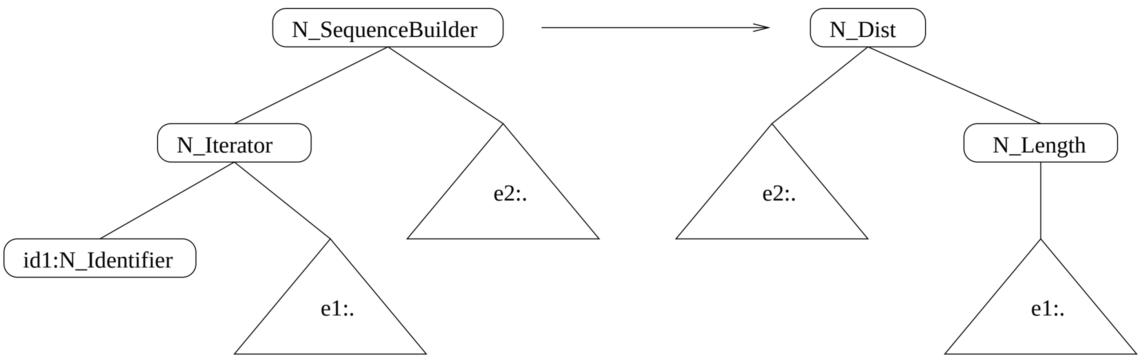
4.4.8 Debugging with KHEPERA

The KHEPERA library tracks changes to the AST throughout the transformation process using the tuple logging algorithms described in Section 3.3.2 (page 45). The tracking is performed, automatically, at the lowest levels of AST manipulation: creation, destruction, copying, and replacement of individual nodes and subtrees. This tracking is transparent, assuming that the programmer always uses the KHEPERA AST-manipulation library, either via direct calls or via the KHEPERA transformation language, to perform all AST transformations. This assumption is reasonable because use of the KHEPERA library is required to maintain AST integrity through the transformation process. Since the programmer does not have to remember to add tracking capabilities to the transformations, the overhead of implementing debugging support in a language processor is greatly reduced.

If e_2 is an **Int** or (e_2 is an **Id** and $e_2 \neq \mathrm{id}_1$), then

$$\overbrace{\texttt{(/}\ \underbrace{\mathrm{id}_1\ \texttt{in}\ e_1}_{\text{N\_Iterator}}\ :\ e_2\ \texttt{/)}}^{\text{N\_SequenceBuilder}} \longrightarrow \texttt{dist(depth=}\ 0, e_2, \texttt{length(}\ e_1\ \texttt{))}$$

(a) Formal Rule 2



(b) AST Transformation

```
rule eliminate_iterator2
{
    match (this:N_SequenceBuilder
           (N_Iterator id1:N_Identifier e1:.) e2:.)
    when (tre_id(e2) == N_Integer
          || (tre_id(e2) == N_Identifier
              && tre_symbol(id1) != tre_symbol(e2)))
    build new with (N_Dist e2 (N_Size e1))
    replace this with new
}
```

(c) KHEPERA Rule

Figure 4.11: Another Simple Transformation Rule (Rule 2)

$$\overbrace{(/\ \underbrace{\mathrm{id}_1\ \texttt{in}\ e_0}_{\text{N\_Iterator}}\ :\ f(\texttt{depth=}\ prime, e_1, \ldots, e_n)\ /)}^{\text{N\_SequenceBuilder}}$$

$$\longrightarrow$$

$$f(\texttt{depth=}\ prime + 1, (/\ \mathrm{id}_1\ \texttt{in}\ e_0\ :\ e_1\ /), \ldots,$$
$$(/\ \mathrm{id}_1\ \texttt{in}\ e_0\ :\ e_n\ /))$$

(a) Formal Rule 3

```
rule dp_func_call
{
    match (this:N_SequenceBuilder
           iter:N_Iterator
           (f:N_Call
            fn:N_Identifier
            plist:N_ExpressionList))

    build newPlist with (N_ExpressionList)
    children plist {
        match (p:.)
        build next with (N_SequenceBuilder
                         iter p)
        do { tre_append(newPlist, next); }
    }

    build call with (N_Call fn newPlist)
    delete newPlist
    do { call->prime = f->prime + 1; }
    replace this with call
}
```

(b) Khepera Rule

Figure 4.12: Iterator Distribution Rule (Rule 3)

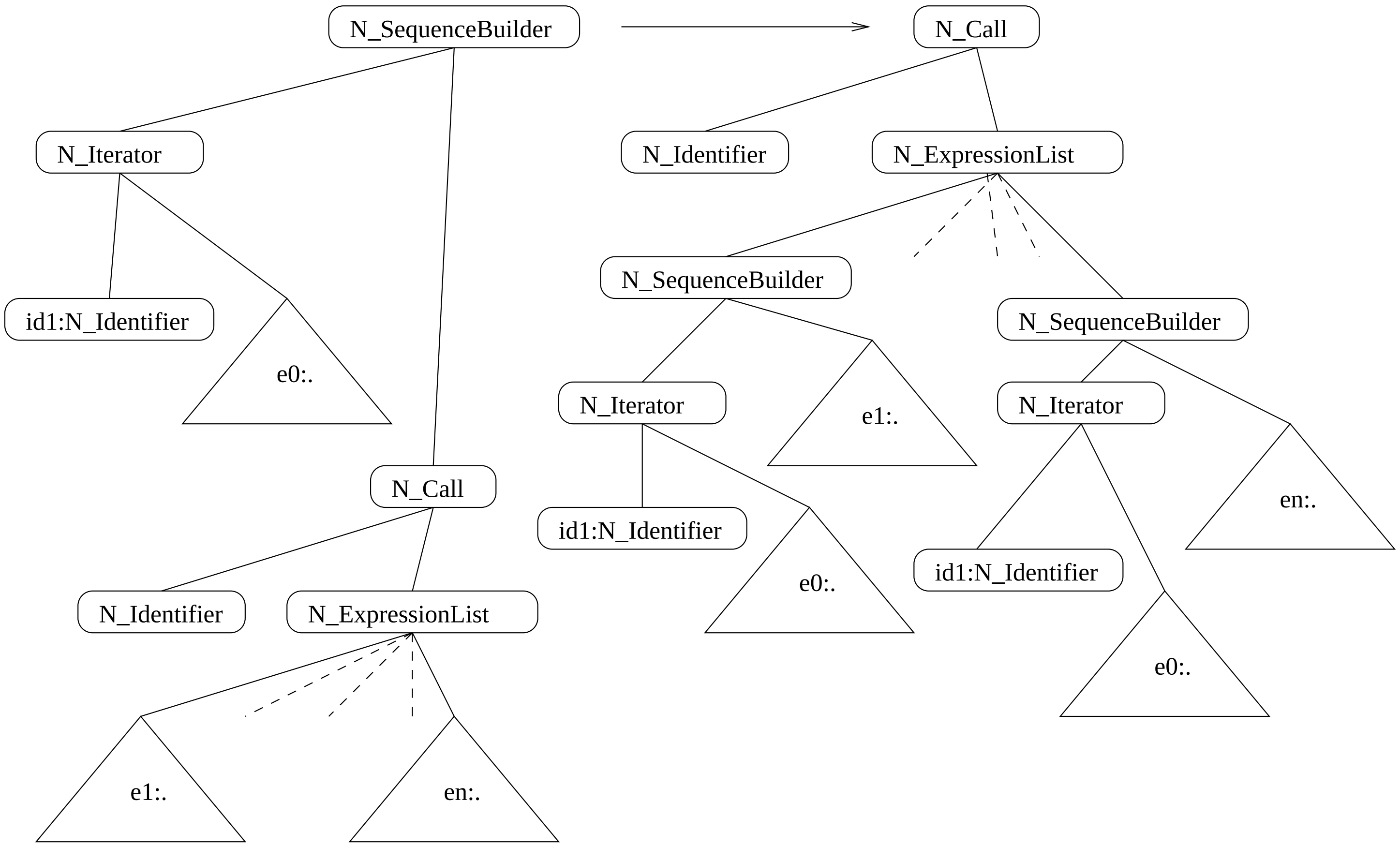


Figure 4.13: AST Transformation for Iterator Distribution Rule (Rule 3)

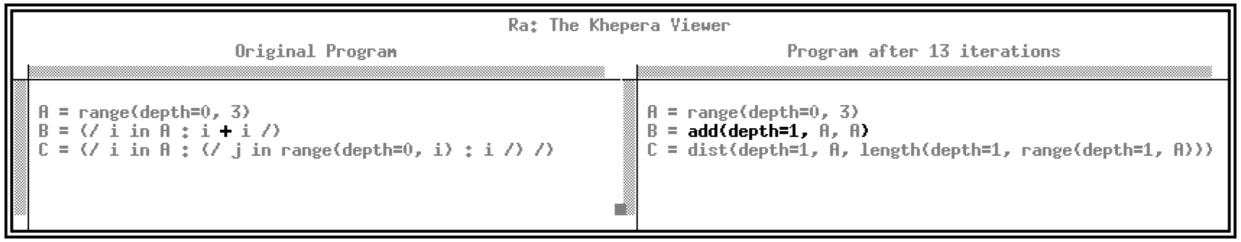
The tracking algorithms associate the tree being transformed (T_i), the transformation rule (τ) being applied, and the specific changes made to the AST. This information can then be analyzed to answer queries about the transformation process. For example, the implementor of the experimental language may have identified two intermediate ASTs, T_i and T_{i+1}, and may ask for a summary of the changes between these two ASTs.

On a more sophisticated level, the user may identify a node in the input program and request that a breakpoint be placed in the program output. An example of this is shown in Figure 4.14a. Here, the user clicked on the scalar `+` node in the left window. In the right window, the generated program, after 13 transformations have been applied, is displayed, showing that the breakpoint should be set on the call to the vector `add` function. The breakpoint was set using the **Track-Breakpoint** algorithm from Section 3.4.3 (page 52).

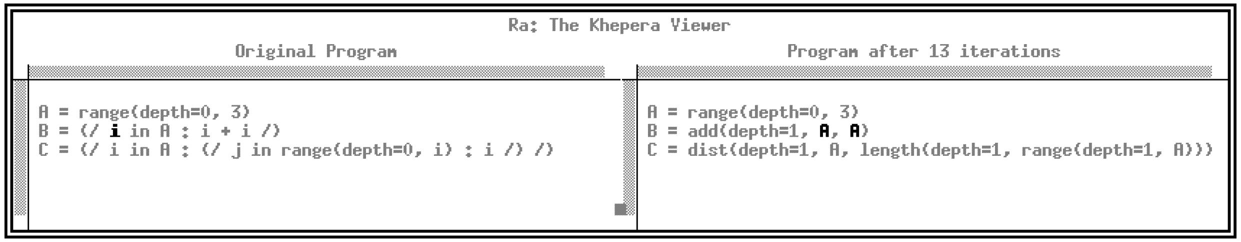
At this point, the user could navigate backward and forward among the transformations, viewing the particular intermediate ASTs that were involved in transforming the original `+` into the call to `add`. These intermediate trees are found using the **Find-Next** algorithm, described in Section 3.5.2.2 (page 59). The ability to navigate among these views is unique to the KHEPERA system and helps the user to understand how the transformations changed the original program. This is especially useful when many transformations are composed.

The tracking algorithms can also be used to understand relationships between variables in the original and transformed programs. For example, in Figure 4.14b, the user has selected an iterator variable `i` that was removed from the final transformed output. In this case, both occurrences of `A` are marked in the final output, showing that these vectors correspond, in some way, to the use of the scalar `i` in the original input. This feature uses the **Track-Breakpoint** algorithm from Section 3.4.3 (page 52).

In addition to the "forward" tracking, described here, KHEPERA also supports reverse tracking, which can be used to determine the current execution point in source terms, or to map a compile or run-time error back to the input source, using the **Track-Execution-Point** algorithm from Section 3.4.4 (page 55).



(a) Example 1: Tracking +



(b) Example 2: Tracking i

Figure 4.14: Debugging with KHEPERA

4.5 Fast Tree Traversal

When the translator attempts to apply rules to the AST, the AST is traversed in either preorder or postorder, with each node examined to see if the current rule matches the subtree rooted at that node. If the rule matches, it is applied, and the traversal of the AST continues from that point. When the traversal finishes, if the rule was applied at least once, then the traversal is repeated and the same rule is matched and applied to the tree again. If the rule did not match during a tree traversal, then another rule is selected, and another tree traversal is performed. This general algorithm for applying a rule τ to a tree T is described in Figure 4.15 for the postorder traversal (the boxes highlight the differences between this algorithm and the **FastApplyRulePostorder** algorithm shown in Figure 4.17 and discussed below).

Depending on the set of rules being applied, many of the details just outlined can be changed. For example, the application of one rule might trigger the application of another rule at the current node, or a rule might not be repeatedly applied to the AST before another rule is selected. However, the general idea is that, whenever a rule is selected, *every* node in the tree is examined to determine if the rule matches, and the rule is applied to those nodes that match. This means that a great deal of matching code is being executed (the first part of Figure 4.10, for example) at every node, but the body of the rule is executed only at nodes which match.

In Section 4.5.1, empirical timing data from a prototype Proteus-to-C translator is presented, showing that the matching operations can be responsible for a significant percentage of total execution time. In Section 4.5.2, observations about the rule matching problem are outlined and a general overview of a method for increasing the performance of the match search is proposed. Section 4.5.3 presents an evolution of several algorithms, the last of which provides best performance for a set of examples. Section 4.5.3.7 presents an analysis of the worst case performance of the algorithms, and empirical data showing that the average or expected performance is better than the worst case performance.

4.5.1 Problem

The prototype Proteus-to-C translator was implemented using C and Sorcerer [Parr 1997]. The execution profile of this translator indicates that about half the run time is spent searching for tree matches. Representative results are shown in

Program	Platform	Tree-Traversal Time (%)	Total Execution Time (seconds)
`qsort`[a]	SPARC[b]	53	100
	Pentium[c]	43	13
`trins`[d]	SPARC	55	339
	Pentium	40	48

Table 4.1: Prototype PROTEUS-to-C Translator Performance

[a] `qsort` performs a simple quicksort [Cormen et al. 1991, Chapter 8] of a sequence of numbers. The program is about 20 lines of PROTEUS, and is transformed into about 475 lines of C code.

[b] The SPARC machine is a SPARCstation-10 with a 125MHz HyperSparc CPU upgrade and 96MB of main store, running SunOS 4.1.4.

[c] The Pentium machine is a 133MHz Pentium with 32MB main store running Linux 2.0.27.

[d] `trins` implements a dynamic programming solution to the triangular solitaire game. The program is about 150 lines of PROTEUS, and is transformed into about 2500 lines of C.

Table 4.1.[2]

These data also point to a problem with using Sorcerer for tree matching. For common tasks (e.g., compiling C code), the SPARC machine used in this example usually performs at about 75% of the processing speed of the Pentium machine (not the 10–20% suggested by the data in Table 4.1). The dramatic difference in speed noted in this example can be attributed to the use of `setjmp/longjmp` by Sorcerer.[3] This mechanism is used to backtrack after partial tree matches. The jumping mechanism is probably expensive on the SPARC processor because of the sliding register windows.

4.5.2 Observations

Most of the rules that are applied in a typical set of transformations match a tree that is rooted at a specific type of node. For example, a transformation rule that assigns symbol table entries to identifiers need only match "identifier" nodes. Similarly, most of the code-restructuring transformations that are used for the flattening of nested-data parallelism match a specific type of syntactic construct [Palmer 1996]. Other rules, that initially appear to require a generic match, can be re-written to match a tree rooted at a specific type of node. For example, CSE requires matching all operator

[2] SunOS is a registered trademark of Sun Microsystems, Inc.; SPARC is a registered trademark of SPARC International, Inc.; Linux is a registered trademark of Linus Torvalds; and Pentium is a registered trademark of Intel Corporation.

[3] Version 1.00B6 (March 1994) was used for these tests. More recent versions of Sorcerer exist, but they also appear to use the `setjmp/longjmp` mechanism.

nodes, but can be re-written as several more specific rules that match each type of operator (e.g., "add", "multiply"). This increases the amount of transformation code but, as discussed in the next section, this re-writing can dramatically decrease the amount of time spent searching for successful matches. Further, simple extensions to the KHEPERA transformation language can make expansion of these sorts of rules automatic, thereby reducing programmer overhead.

Nyland [1994] first suggested that an improvement in tree pattern matching could be realized if lists are maintained of each type of node, and only one list is traversed when searching for tree matches. For example, if the rule matches a tree of the form `(Plus Identifier Constant)`, then examining the list of `Plus` nodes is sufficient to find all of the matches in the tree. However, implementing a "fast" tree walker is *not* as simple as maintaining unordered lists of nodes by type. Instead, the lists must be maintained in the order in which the nodes would have been visited if a full preorder or postorder traversal of the AST was performed.

When used for transformation application, a "fast" tree walker must generate identical results when compared with a standard tree walker. Since transformations may assume a preorder or postorder transversal of the tree, the "fast" tree walkers must preserve this ordering. Therefore, the lists of nodes must be ordered. Data structures which permit rapid insertion into an ordered list and inorder traversal may be suitable for storing these lists.

Without loss of generality, I will assume that the desired traversal order is postorder, and will only discuss this case. However, similar observations and solutions are also possible for other traversal orders. Figure 4.16b shows a tree labeled with consecutive integers such that visiting each node in increasing integer order is the same as visiting each node in postorder. Figure 4.16c shows lists for each node type. Visiting the nodes of a specific type in increasing order in the list is equivalent to traversing the tree in postorder and only visiting the the nodes of the specific type. An algorithm, similar to **ApplyRulePostorder** (Figure 4.15), is shown in Figure 4.17, with the key differences denoted with boxex.

Since the transformations make changes to the tree during the traversal, the lists must be updated on-the-fly for each tree alteration. Therefore, the lists must be stored in data structures that can be updated in the middle of a walk through the data structure. This rules out the use of red-black trees, 2-3-4 trees, AVL trees, or other "balanced" binary tree structures: an insertion into one of these structures can result in a rotation such that a stack-based traversal which is in progress at the time of the insertion is perturbed. Figure 4.18 shows an example of this problem [Cormen et al. 1991, p. 266]. Assume that an inorder traversal of the tree is being performed, and that x has been matched by a transformation rule that performs an operation requiring a rotation of the red-black tree. Before the rotation, subtree β will be traversed next, and then the walk will proceed to y, since y is currently on the stack. However, after the insertion, node y is the next node: β, y, and γ will be traversed, and then the stack will return the walk to y. This problem could be overcome by using a doubly-linked red-black tree, and avoiding the use of a stack during red-black tree traversal. However, simpler data structures exist which do not require special threading for traversal in the face of insertions and deletions.

An unbalanced binary tree meets the requirements for the list-containing data structure, but would likely become degenerate during the transformation process, making insertions into the list very expensive. Skip lists [Pugh 1990b] meet the requirements and have probabilistic $O(\lg n)$ amortized performance for searches, insertions, and deletions. Further, because of their simple implementation, skip lists may have a smaller constant factor than a comparable balanced binary tree implementation.

Other solutions to this problem are possible. However, any solution must be constrained by several implementation considerations:

- if the nodes are labeled, the labels must fit in a reasonable number of bits (since the tree can be degenerate, using one bit per tree level is not an implementation option), and

- the solution must not dramatically increase the memory required to store a node (since there may be hundreds of different node types in a typical application (e.g., the Proteus-to-C translator), threading all of the node lists through each node is not an implementation option).

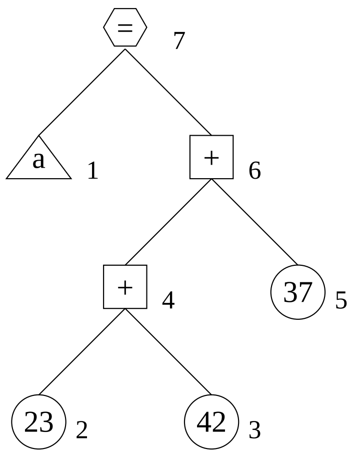
Because of these constraints, other solutions that I have considered (but have not implemented) appear to require at least as much work as the algorithms described in the next section.

ApplyRulePostorder($\boldsymbol{T}, \boldsymbol{\tau}$) returns $\boldsymbol{T'}$
Input:
T, the current tree
τ, the transformation rule being applied
Output:
T', the final transformed tree
Notes:
Match is a function that returns true if the rule "matches" at the specified subtree. This function performs the same actions as the **match** and **when** commands in the KHEPERA language, described with examples in Section 4.4.3.
Apply is a function that applies the body of the rule to the specified subtree. This function performs the same actions as the body of a rule written in the KHEPERA language, applying changes to the tree, as outlined in Section 3.3.1.
First-Postorder is a function that returns first node that should be visited for a postorder traversal of T.
Next-Postorder is a function that returns the next node in the tree in postorder. The walk is always continued from the current node, so that the effects of AST changes can be taken into account.
Algorithm
do
$f \leftarrow$ **false**
$n \leftarrow \boxed{\textbf{First-Postorder}(\boldsymbol{T})}$
while $n \neq \varnothing$ **do**
if Match(τ, n) **then**
Apply(τ, n)
$f \leftarrow$ **true**
$n \leftarrow \boxed{\textbf{Next-Postorder}(\boldsymbol{n})}$
while $f =$ **true**
End of ApplyRulePostorder

Figure 4.15: ApplyRulePostorder Algorithm

$$a = (23 + 42) + 37$$

(a) Expression



(b) AST Labelled in Postorder

Node Type		List of Labels
□	+	4, 6
○	Integer	2, 3, 5

(c) Per-Node Lists

Figure 4.16: Postorder Labelling of Tree and Corresponding Per-Node Lists

FastApplyRulePostorder($\boldsymbol{T, \tau, S_t}$) returns $\boldsymbol{T'}$

Input:

T, the current tree

τ, the transformation rule being applied, which matches a subtree rooted at a node of type t

S_t, the list of nodes of type t arranged such that iterating over all of the nodes in S is equivalent to a postorder traversal of these nodes in T

Output:

T', the final transformed tree

Notes:

Match is a function that returns true if the rule "matches" at the specified subtree. This function performs the same actions as the **match** and **when** commands in the KHEPERA language, described with examples in Section 4.4.3.

Apply is a function that applies the body of the rule to the specified subtree. This function performs the same actions as the body of a rule written in the KHEPERA language, applying changes to the tree, as outlined in Section 3.3.1.

First-SkipList returns the first node in the skiplist for the type of node that is the root of the subtree matched by τ.

Next-SkipList returns the next node in the skiplist for the same type of node as n. The walk is always continued from the current node, so that the effects of AST changes can be taken into account.

Algorithm

do
  $f \leftarrow$ **false**
  $n \leftarrow \boxed{\textbf{First-SkipList}(\boldsymbol{\tau})}$
  while $n \neq \varnothing$ **do**
    if Match(τ, n) **then**
      Apply(τ, n)
      $f \leftarrow$ **true**
    $n \leftarrow \boxed{\textbf{Next-SkipList}(\boldsymbol{n})}$
while $f =$ **true**

End of FastApplyRulePostorder

Figure 4.17: FastApplyRulePostorder Algorithm

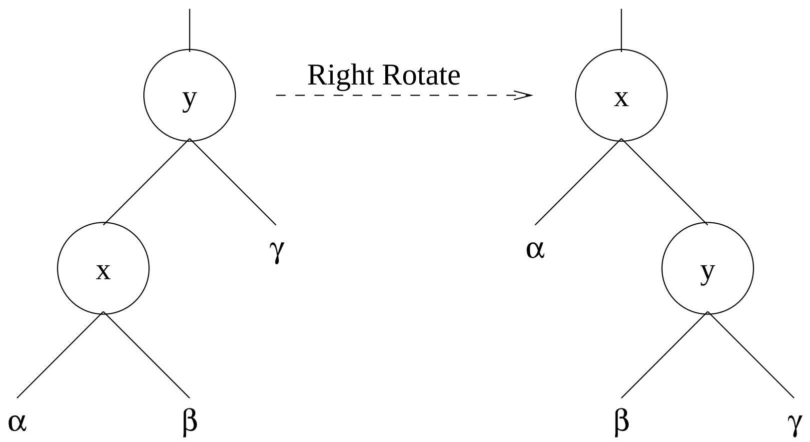


Figure 4.18: Rotation on a Red-Black Tree

4.5.3 Fast Tree-Traversal Algorithms

The algorithms presented here assume skip lists are being used for the implementation of the node lists. This simplifies some of the language necessary to describe the algorithms and the interaction between the AST data structure and the data structure which is used to store the node lists. However, skip lists do not have to be used—any data structure which allows a first-to-last traversal in the face of insertions and deletions can be used.

Note that if nodes are deleted from the AST, they can simply be deleted from the skip lists, and a postorder walk can still be performed on the AST. Hence, the following algorithms only address insertions of new subtrees into the AST. The difficulty is to design an algorithm that maintains the skip lists but which does not incur greater overhead than would a simple postorder traversal of the whole AST. **Algorithm0** is used within the **Apply** algorithm shown in **ApplyRulePostorder**. The other algorithms are used within the **Apply** algorithm shown in **FastApplyRulePostorder**.

The five algorithms presented below describe the successive improvements of a design for a suitable algorithm for fast tree traversal. Algorithm 0 is the base case, and Algorithm 4 is the final suggested algorithm. Without loss of generality, these algorithms all assume that the desired tree-traversal sequence is postorder. If a preorder traversal is desired, the labelling algorithms require slight modifications. Since postorder is a commonly used traversal order for transformation rule application, all of the empirical data collection assumes that fast matching is available for postorder traversals, but not for preorder traversals.

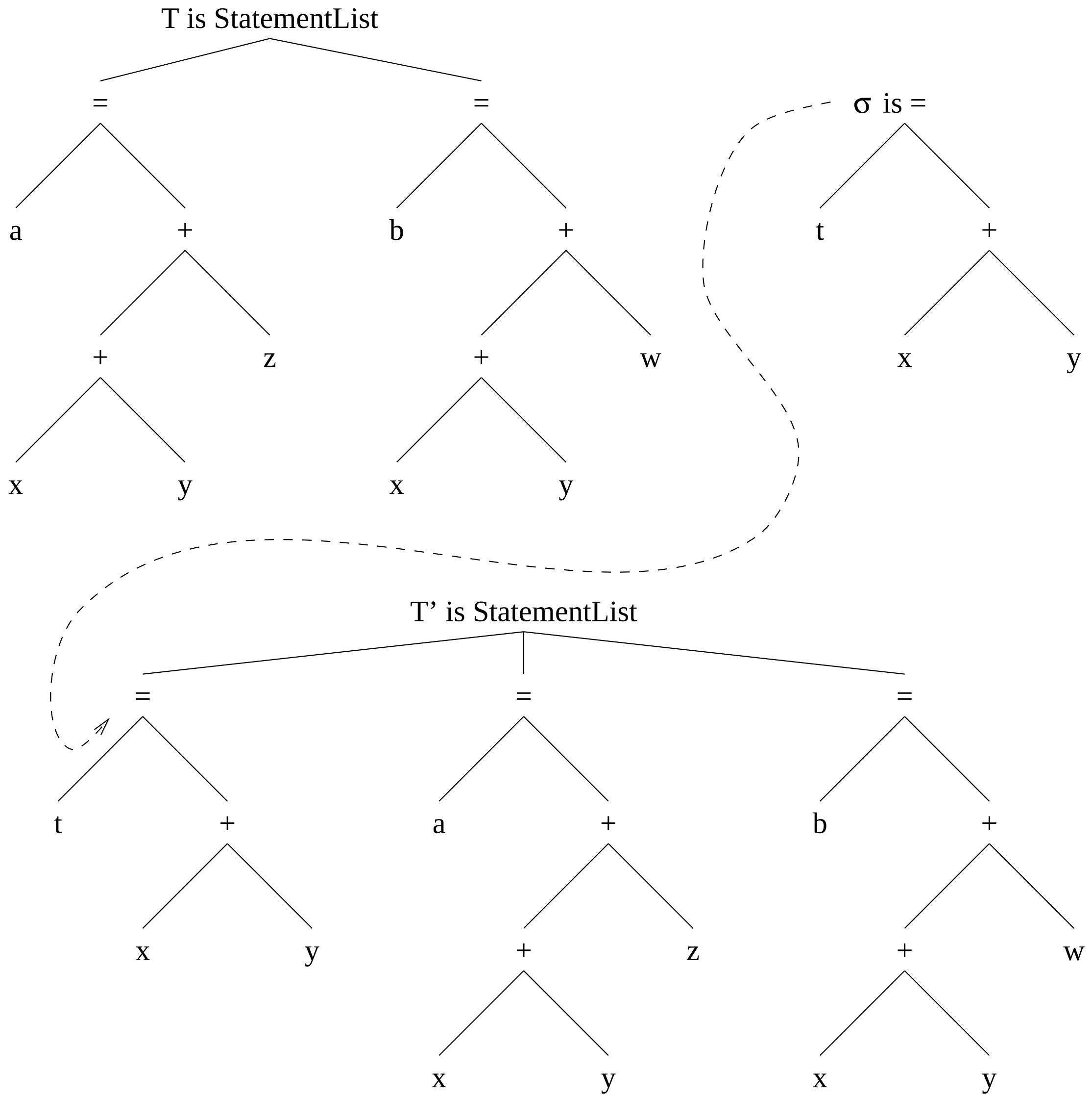
The example transformation shown in Figure 4.19a, the initial step of a CSE algorithm, will be used throughout this discussion to illustrate how the various algorithms manipulate the AST and associated data structures. Note that the next step in CSE would be to replace the $x + y$ expressions in the assignments to a and b. This step involves two tree replacements, and needlessly complicates this example.

4.5.3.1 Notation

For this discussion, T and T' represent complete ASTs, σ is the subtree being inserted at some point on the AST, and n refers to a node. Nodes have attributes, as shown in Table 4.2. The '.' operator is used to access an attribute value, so n.l refers to the minimum label range for node n. For each node *type*, a global skip list exists, S_{type}. When **Apply-Need-Based-Labels** is introduced, there will be constants for

```
a = (x + y) + z          ⟶          t = x + y
b = (x + y) + w                      a = (x + y) + z
                                     b = (x + y) + w
```

(a) Initial Step for CSE



(b) Insertion on AST

Figure 4.19: Example of Algorithm 0

Attribute Name	Meaning	Type
type	the type of the node	integer
parent	the parent of the node	node
child	the first child of the node	node
left-sibling	the left-hand sibling of the node	node
right-sibling	the right-hand child of the node	node
label	the label of the node	integer
left-need	the label needs for the left-hand subtree	integer
right-need	the label needs for the right-hand subtree	integer
need	left-need + right-need	integer
l	minimum label range for left-hand subtree	integer
r	maximum label range for right-hand subtree	integer

Table 4.2: Node Attributes

the minimal label value, *min-label*, and the maximal label value, *max-label*, usually selected to be zero and the maximum representable integer. For purposes of the examples, however, *max-label* is defined to be 100.

The tree is an n-ary tree, with an underlying tilted binary tree representation [Knuth 1973]. On the binary tree representation:

- the left-hand child is the **child** attribute on the n-ary tree,
- the right-hand child is the **right-sibling** attribute on the n-ary tree, and
- the parent is the **left-sibling** attribute on the n-ary tree, or, if the **left-sibling** attribute is $\varnothing$, the **parent** attribute.

An example of an n-ary tree and the equivalent binary tree is shown in Figure 4.20. Some of the algorithms described below depend the ability to traverse the n-ary tree as if it were a tilted binary tree. This is the underlying implementation used in KHEPERA, but the ability to perform these sorts of traversals is not dependent on the underlying representation: any n-ary tree can be traversed as if it were a binary tree.

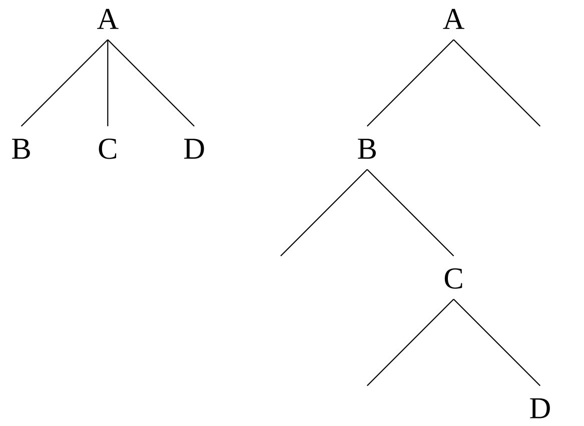


Figure 4.20: An n-ary Tree and an Equivalent Binary Tree

4.5.3.2 Algorithm 0

Algorithm 0, shown in Figure 4.21, does not use skip lists or attempt to perform any "fast" walks. An example of the input and output for this algorithm is shown in Figure 4.19.

Algorithm0(T, σ) returns T'
Input:
    T, the current tree
    σ, the subtree being inserted into the tree
Output:
    T', the tree T with σ inserted
Algorithm
    $T' \leftarrow T$ with σ inserted
End of Algorithm0

Figure 4.21: Algorithm 0

Algorithm 0 is the reference case for all of the other algorithms, both in terms of performance and in terms of correctness. The KHEPERA debugging tracking features can be used to log all AST manipulations to a file. The output of this logging should be identical regardless of which algorithm is being used.

4.5.3.3 Algorithm 1

Algorithm 1 is shown in Figure 4.22. Algorithm 1 uses **Apply-Consecutive-Labels**, shown in Figure 4.23a, and **Skip-List-Insert**, shown in Figure 4.23b.

Algorithm1($\boldsymbol{T, \sigma}$) returns $\boldsymbol{T'}$
Input:
    T, the current tree
    σ, the subtree being inserted into the tree
Output:
    T', the tree T with σ inserted and attributes updated
Algorithm:
    $T' \leftarrow T$ with σ inserted
    Apply-Consecutive-Labels(T')
    Skip-List-Insert(σ)
End of Algorithm1

Figure 4.22: Algorithm 1

Whenever a subtree σ is inserted into the AST, the whole AST is traversed to apply labels, and then the nodes on σ are inserted into the appropriate skip lists. Since an insertion of subtree σ does not change the *relative* ordering of any of the other nodes in the AST, nodes do not have to be deleted from the skip lists before the relabeling step (although, *during* the relabeling step, the skip lists may be in an undefined state—this is not a problem, since the lists are not accessed during this step).

In Figure 4.24a, the tree and skip lists are shown immediately after σ is inserted into T. Figure 4.24b shows the tree and skip lists after **Apply-Consecutive-Labels** has executed—note that the relabelling step also updates the labels in the skip list. Figure 4.24c shows the skip lists after **Skip-List-Insert** has executed (the labels on the tree are the same as in Figure 4.24b).

This algorithm minimizes skip list operations, but still requires a complete traversal of the AST for the relabeling step. Unfortunately, the relabeling step cannot be completely avoided without using labels which have at least one bit for each tree level. The remaining algorithms will try to minimize the amount of relabeling needed for many (but not all) insertions.

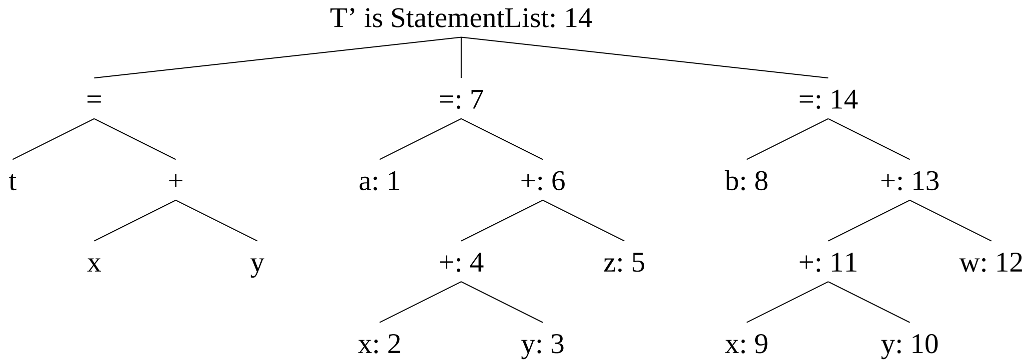
Apply-Consecutive-Labels($\boldsymbol{T}$)
Input:
    T, the current tree
Output:
    T, the current tree with label attributes updated
Notes:
    First-Postorder is a function that returns first node that should be visited for a postorder traversal of T.
    Next-Postorder is a function that returns the next node in the tree in postorder.
Algorithm:
    $c \leftarrow 0$
    $n \leftarrow$ **Root($\boldsymbol{T}$)**
    while $n \neq \varnothing$ **do**
        n.label $\leftarrow c$
        $c \leftarrow c + 1$
        $n \leftarrow$ **Next-Postorder($\boldsymbol{n}$)**
End of Apply-Consecutive-Labels

(a) Apply-Consecutive-Labels Algorithm

Skip-List-Insert($\boldsymbol{\sigma}$)
Input:
    σ, subtree
Output:
    Skip lists updated. For each node *type*, there is a global skip list, S_{type}.
    The type of a node is accessible via the *type* attribute.
Notes:
    First-Postorder is a function that returns first node that should be visited for a postorder traversal of T.
    Next-Postorder is a function that returns the next node in the tree in postorder.
Algorithm:
    $n \leftarrow$ **First-Postorder($\boldsymbol{T}$)**
    while $n \neq \varnothing$ **do**
        Insert n into skip list $S_{n.\text{type}}$
        $n \leftarrow$ **Next-Postorder($\boldsymbol{n}$)**
End of Skip-List-Insert

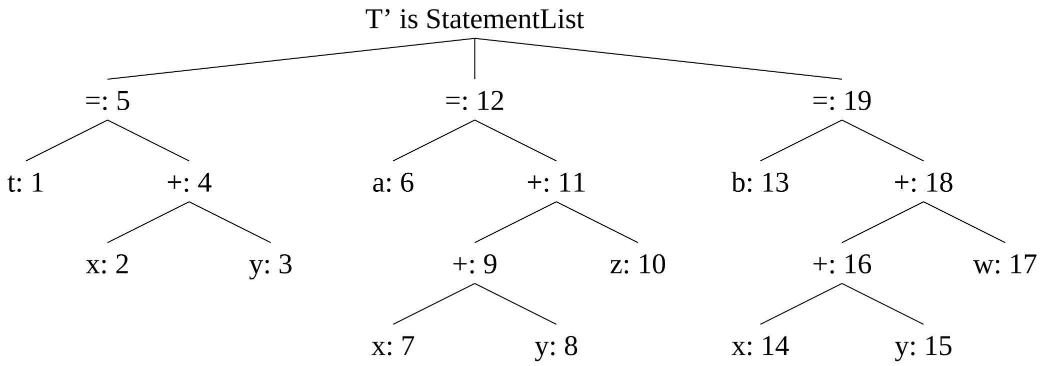
(b) Skip-List-Insert Algorithm

Figure 4.23: Apply-Consecutive-Labels and Skip-List-Insert Algorithms



$$\begin{aligned} = &: 7, 14 \\ + &: 4, 6, 11, 13 \\ \mathbf{id} &: 1, 2, 3, 5, 8, 9, 10, 12 \end{aligned}$$

(a) Tree and Skip Lists After σ inserted into T



$$\begin{aligned} = &: 12, 19 \\ + &: 9, 11, 16, 18 \\ \mathbf{id} &: 6, 7, 8, 10, 13, 14, 15, 17 \end{aligned}$$

(b) Tree and Skip Lists After Apply-Consecutive-Labels

$$\begin{aligned} = &: 5, 12, 19 \\ + &: 4, 9, 11, 16, 18 \\ \mathbf{id} &: 1, 2, 3, 6, 7, 8, 10, 13, 14, 15, 17 \end{aligned}$$

(c) Skip Lists After Skip-List-Insert

Figure 4.24: Example of Algorithm 1

4.5.3.4 Algorithm 2

Algorithm 2, shown in Figure 4.25a, relabels every node on the tree, inserting all the new nodes on σ into the skip lists. Since an insertion does not change the ordering of the nodes in the AST, the nodes do not have to be removed from the skip lists—they just have to be relabeled.

The **Compute-Needs** function performs a *reverse postorder* walk on the *binary tree* representation (not the n-ary representation) of the AST T, keeping track of the total number of "left-hand" and "right-hand" children under the node. Note that a reverse postorder traversal of a binary tree is like a regular postorder traversal, except that the right-hand child is visited *before* the left-hand child.

For each node n visited in the reverse postorder traversal of the binary representation of T, **Compute-Needs** evaluates the algorithm shown in Figure 4.25b.

Figure 4.26a shows the tree after σ has been inserted and **Compute-Needs** has been run. The two numbers on each node are the left and right "needs" for that node—intuitively, the total number of children under that node on the left and right sides. The dotted lines point to the left (downward) and right (rightward) children for each node, when traversed using the reverse postorder traversal of the *binary tree* representation of the n-ary AST.

The **Apply-Need-Based-Labels** function performs a preorder walk of the AST T, computing unique labels from limits set in the parent. For each node, n, visited in the preorder traversal of T, **Apply-Need-Based-Labels** evaluates the algorithm shown in Figure 4.27a. **Apply-Need-Based-Labels** uses **Apply-Limits**, shown in Figure 4.27b.

Figure 4.26b shows the tree after **Apply-Need-Based-Labels** has been run. The three numbers on each node are the l value, the label, and the r value from the algorithm. For this example, *min-label* $\stackrel{def}{=}$ 0, and *max-label* $\stackrel{def}{=}$ 100. The dotted lines point to the left (downward) and right (rightward) children for each node, when traversed using the reverse postorder traversal. Intuitively, this algorithm is taking the available labels (in this case, $0, 1, 2, \ldots, 100$), and partitioning them among the children in a manner weighted by the number of children in the left and right subtrees (the "needs" computed above). The effect is that:

1. There are always enough labels to label the nodes that exist on the current tree.

2. There are extra labels available that might be used when a small tree is inserted. The following algorithms explore ways to determine when enough labels exist

for an insertion, and when some or all of the tree must be relabelled before an insertion can take place.

The complex expression in the last line of the algorithm divides up the labels among the nodes, with care that labels can be within one unit of each other, but are never the same.

This algorithm is easier to understand if we remember the n-ary AST is implemented using a tilted binary tree. Our goal is to label the n-ary AST in postorder—this is the equivalent of labeling the tilted binary tree inorder [Knuth 1973, p. 335]. Figure 4.28 shows the tree and skip lists before and after the new labels have been applied, and shows the skip lists after **Skip-List-Insert** has executed.

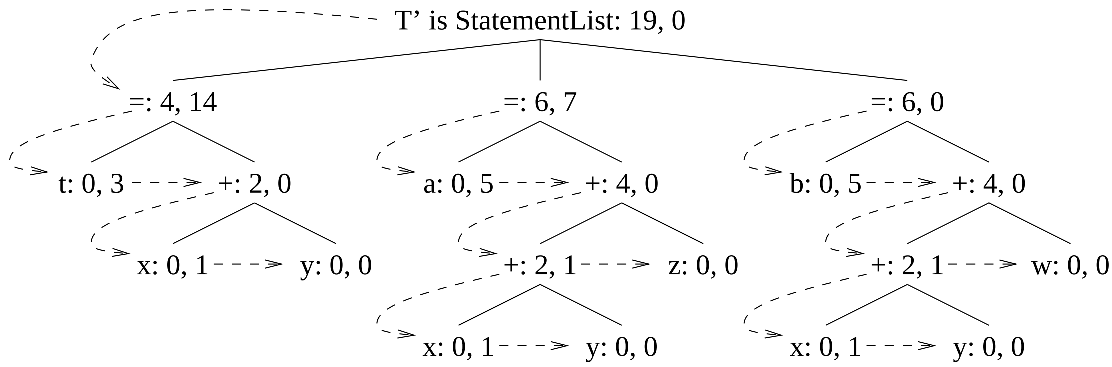
Algorithm2($\boldsymbol{T}, \boldsymbol{\sigma}$) returns $\boldsymbol{T'}$
Input:
    T, the current tree
    σ, the subtree being inserted into the tree
Output:
    T', the tree T with σ inserted and attributes updated
Algorithm:
    $T' \leftarrow T$ with σ inserted
    Compute-Needs(T')
    Apply-Need-Based-Labels(T')
    Skip-List-Insert(σ)
End of Algorithm2

(a) Algorithm 2

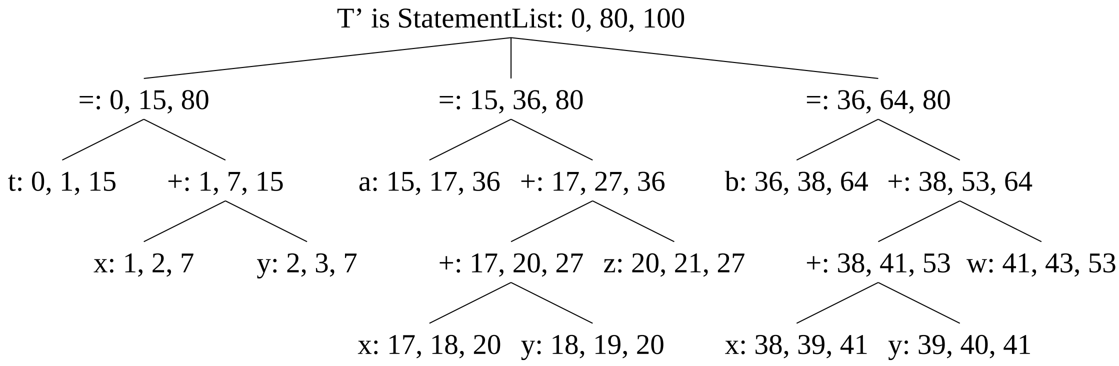
Compute-Needs($\boldsymbol{T}$)
Input:
    T, the current tree
Output:
    T, the tree T, with attributes updated
Notes:
    First-Reverse-Postorder is a function that returns first node that should be visited for a *reverse postorder* traversal of the *binary* representation of T.
    Next-Reverse-Postorder is a function that returns the next node in the *binary* representation of the tree in *reverse postorder*.
Algorithm:
    $n \leftarrow$ **First-Reverse-Postorder($\boldsymbol{T}$)**
    while $n \neq \varnothing$ **do**
        n.left-need $\leftarrow 0$
        n.right-need $\leftarrow 0$
        if n.child $\neq \varnothing$ **then**
            n.left-need $\leftarrow 1 + n$.child.need
        endif
        if n.right-sibling $\neq 0$ **then**
            n.right-need $\leftarrow 1 + n$.right-sibling.need
        endif
        n.need $= n$.left-need $+ n$.right-need
        $n \leftarrow$ **Next-Reverse-Postorder($\boldsymbol{n}$)**
End of Compute-Needs

(b) Compute-Needs Algorithm

Figure 4.25: Algorithm 2 and Compute-Needs



(a) After Compute-Needs



(b) After Apply-Need-Based-Labels

Figure 4.26: Example of the Compute-Needs and Apply-Need-Based-Labels Algorithms

Apply-Need-Based-Labels(T)
Input:
  T, the current tree
Output:
  T, the tree T, with attributes updated
Notes
  First-Preorder is a function that returns first node that should be visited for a preorder traversal of the *binary* representation of T.
  Next-Preorder is a function that returns the next node in the *binary* representation of the tree in preorder.
Algorithm:

```
n ← First-Preorder(T)
while n ≠ ∅ do
    Apply-Limits(n)
    n.label ← n.l + (n.left-need + 1) ⌊(n.r − n.l − 2)/(n.need + 4)⌋
    n ← Next-Preorder(n)
```

End of Apply-Need-Based-Labels

(a) Apply-Need-Based-Labels Algorithm

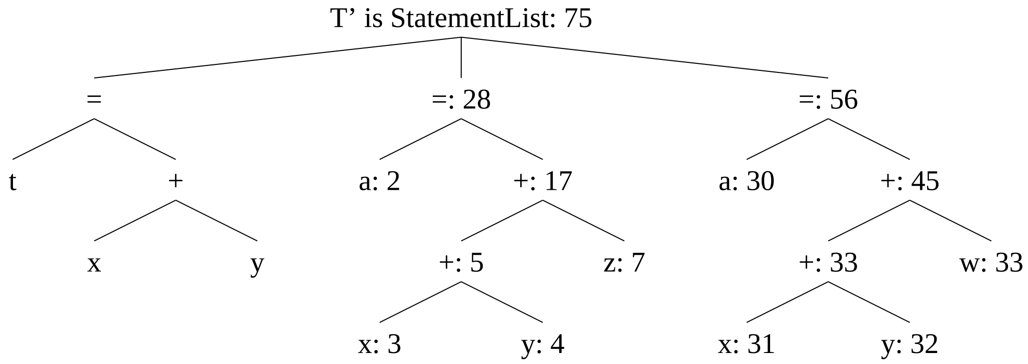
Apply-Limits(n)
Input:
  n, a node
Output:
  n, with limits l and r updated
Algorithm:

```
if n.left-sibling = ∅ then
    if n.parent = ∅ then
        n.l ← min-label
        n.r ← max-label
    else
        n.l ← n.parent.l
        n.r ← n.parent.label
    endif
else
    n.l ← n.left-sibling.label
    n.r ← n.left-sibling.r
endif
```

End of Apply-Limits

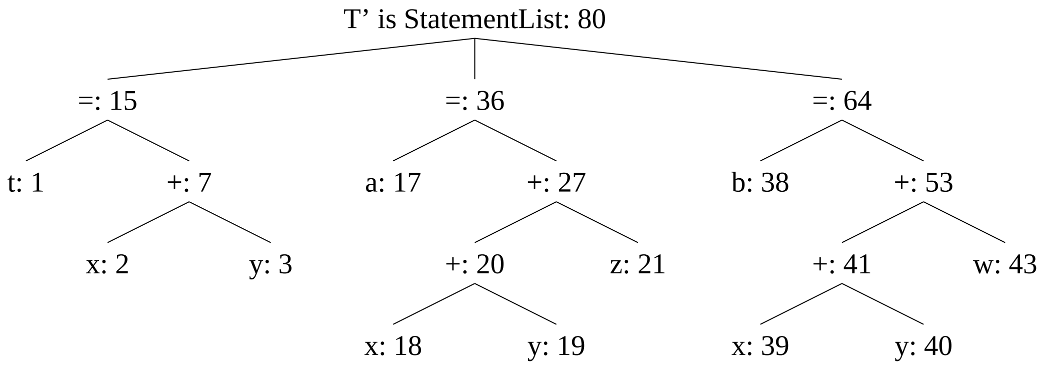
(b) Apply-Limits Algorithm

Figure 4.27: Apply-Need-Based-Labels and Apply-Limits Algorithms



$$\begin{array}{rcl} = & : & 28, 56 \\ + & : & 5, 17, 30, 45 \\ \mathbf{id} & : & 2, 3, 4, 7, 30, 31, 32, 33 \end{array}$$

(a) Tree and Skip Lists After σ inserted into T



$$\begin{array}{rcl} = & : & 36, 64 \\ + & : & 20, 27, 41, 53 \\ \mathbf{id} & : & 17, 18, 19, 21, 38, 39, 40, 43 \end{array}$$

(b) Tree and Skip Lists After Apply-Need-Based-Labels

$$\begin{array}{rcl} = & : & 15, 36, 64 \\ + & : & 7, 20, 27, 41, 53 \\ \mathbf{id} & : & 1, 2, 3, 17, 18, 19, 21, 38, 39, 40, 43 \end{array}$$

(c) Skip Lists After Skip-List-Insert

Figure 4.28: Example of Algorithm 2

4.5.3.5 Algorithm 3

Algorithm 3, shown in Figure 4.29, uses **Binary-Ancestors-Of**, shown in Figure 4.30.

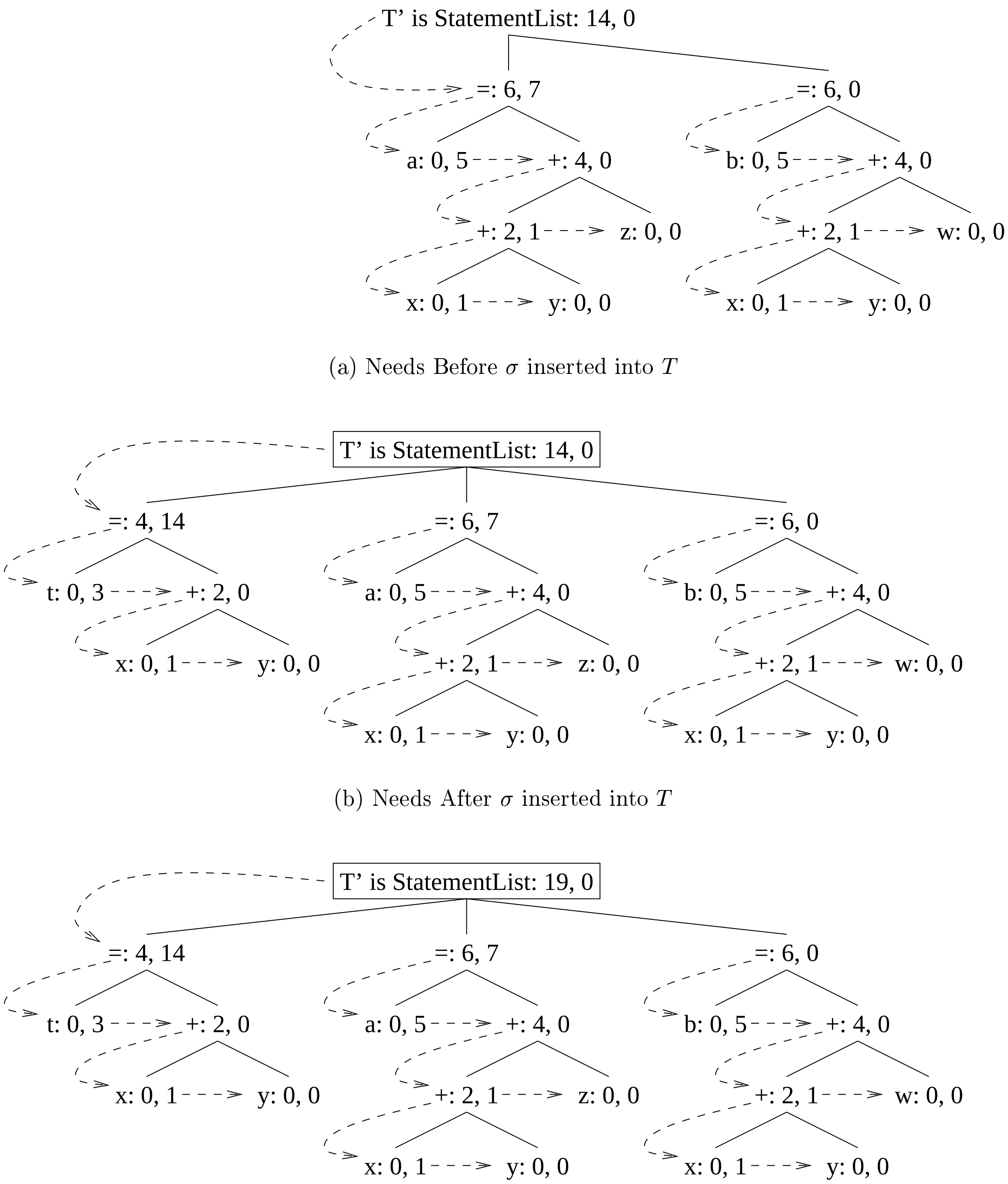
Algorithm3($\boldsymbol{T}, \boldsymbol{\sigma}$) returns $\boldsymbol{T'}$
Input:
  T, the current tree
  σ, the subtree being inserted into the tree
Output:
  T', the tree T with σ inserted and attributes updated
Notes: the parent attribute of the root of T is $\varnothing$
Algorithm:
  $T' \leftarrow T$ with σ inserted
  Compute-Needs(σ)
  for n **in Binary-Ancestors-Of**(σ) **do**
    Compute-Needs(n)
  Apply-Need-Based-Labels(T')
  Skip-List-Insert(σ)
End of Algorithm3

Figure 4.29: Algorithm 3

Since "needs" are computed in postorder, the insertion of σ changes the computed needs only for the ancestors of σ, not for all of the nodes in the tree. So, Algorithm 3 only computes the new needs for these nodes, as shown in Figure 4.31. For this simple example, after the needs are computed, **Apply-Need-Based-Labels** proceeds exactly as in Figure 4.28b, and **Skip-List-Insert** proceeds exactly as in Figure 4.28c.

Binary-Ancestors-Of($\boldsymbol{n}$) returns $\boldsymbol{A}$
Input:
    n, the reference node
Output:
    A, an ordered list containing all of the binary-tree ancestors of n
Algorithm:
    $A \leftarrow \varnothing$
    $p \leftarrow n$
    while $p \neq \varnothing$ **do**
        $A \leftarrow$ **append**(A, p)
        if p.left-sibling $= \varnothing$ **then**
            $p \leftarrow p$.parent
        else
            $p \leftarrow p$.left-sibling
        endif
End of Binary-Ancestors-Of

Figure 4.30: Binary-Ancestors-Of Algorithm



(a) Needs Before σ inserted into T

(b) Needs After σ inserted into T

(c) Needs For Ancestor of σ Updated

Figure 4.31: Example of Algorithm 3

4.5.3.6 Algorithm 4

Algorithm 4, shown in Figure 4.32, demonstrates the reason for computing needs and need-based labels: sometimes it will be possible to insert a new subtree, σ, into the AST and relabel only part of the AST. This partial relabeling is possible because the need-based labels are spread out such that there are often extra labels available to label a newly inserted subtree. This algorithm is more efficient than the others for the PROTEUS-to-C transformations. Because it shares worst-case performance with Algorithm 3, it should always do at least as well as that algorithm.

Algorithm4($\boldsymbol{T, \sigma, T'}$)
Input:
    T, the current tree
    σ, the subtree being inserted into the tree
Output:
    T', the tree T, with σ inserted and attributes updated
Notes: the parent attribute of the root of T is $\varnothing$
Algorithm:
    $T' \leftarrow T$ with σ inserted
    Compute-Needs(σ)
    for n **in Binary-Ancestors-Of**(σ) **do**
        Compute-Needs(n)
    Apply-Limits(σ)
    if σ.l >= σ.r **then**
        /* not enough extra labels exist, so relabel whole tree */
        Apply-Need-Based-Labels(T')
    else
        /* enough extra labels exist */
        Apply-Need-Based-Labels(σ)
    endif
    Skip-List-Insert(σ)
End of Algorithm4

Figure 4.32: Algorithm 4

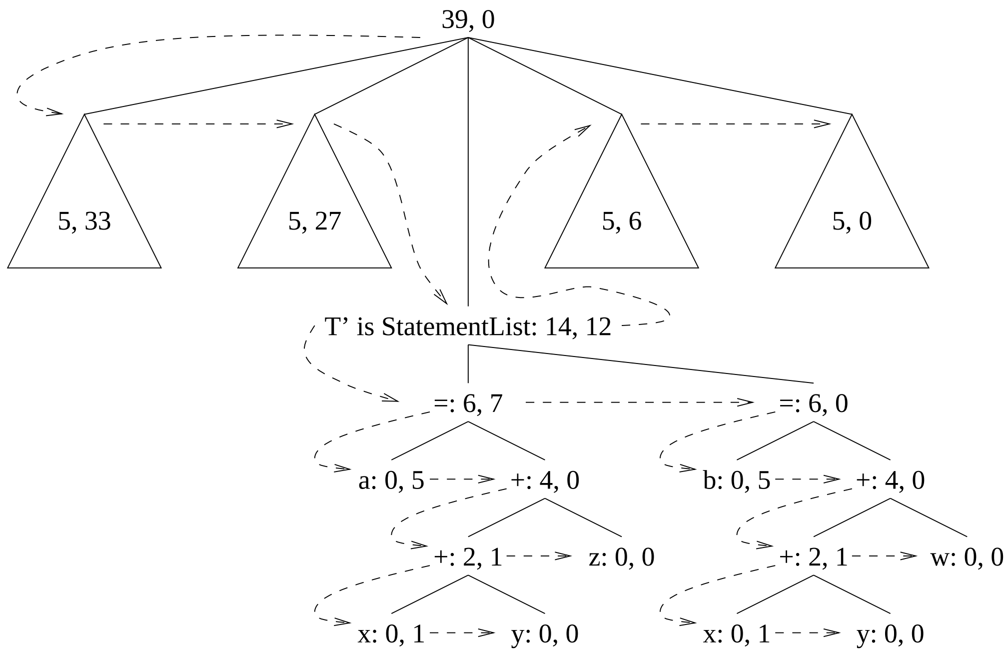
Unfortunately, the example I have been using is too small to demonstrate the differences between Algorithm 3 and Algorithm 4. So, I will extend the example AST with additional ancestors (perhaps in the form of a **case** or **switch** statement), as shown in Figure 4.33a with need attributes. Figure 4.33b shows the tree after σ has been inserted and the needs of the ancestors have been updated. Updates have been marked with boxes.

When **Apply-Need-Based-Labels** is executed, only the inserted tree and the rightward siblings of this tree will be relabeled: none of the leftward siblings or ancestors are relabelled.

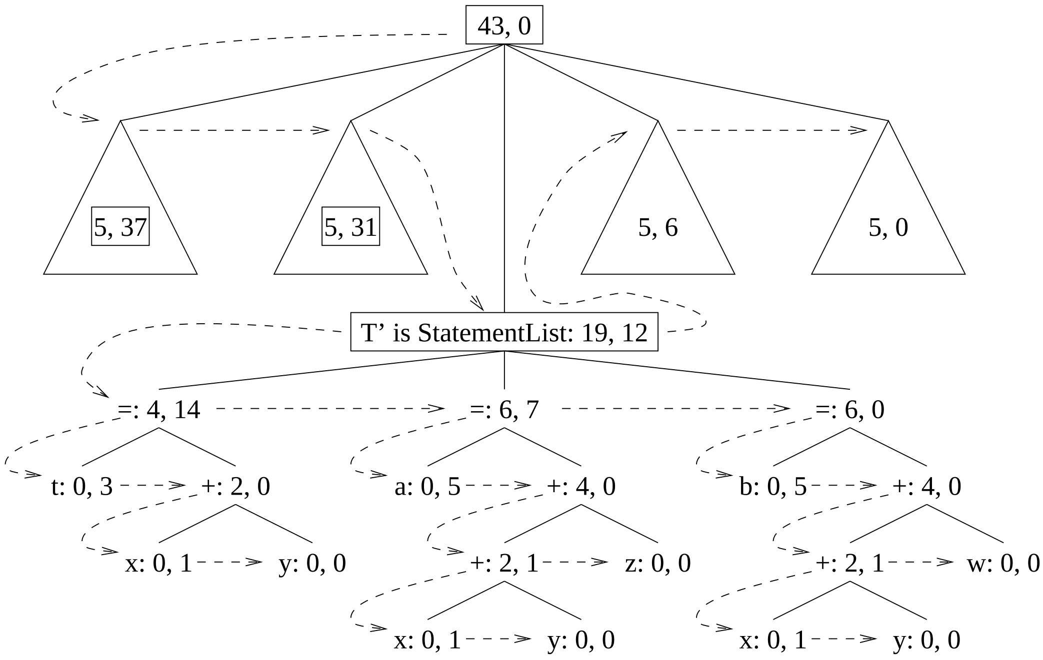
Algorithm 4 can be improved in two ways:

1. When computing the label needs of the ancestors, it may not be necessary to traverse all the way to the top of the AST. A previous deletion (which did not update the label needs values) may have created extra unused labels.

2. The need-based labels do not have to be applied to all siblings of σ. Instead, it may be possible (if enough extra labels are available), to modify the ranges of the rightward sibling, and to relabel only σ and one rightward sibling of σ. Currently, σ and all rightward siblings of σ are modified.



(a) Needs Before σ inserted into T



(b) Needs After σ inserted into T

Figure 4.33: Example of Algorithm 4

4.5.3.7 Analysis

If each node visit is counted as a unit of work, then the worst-case performance of Algorithms 1 through 4 is dominated by the size of the final tree T', and their asymptotic work is $O(\overline{T'})$ (an overline is an operator which returns the number of nodes in a tree, so $\overline{T'}$ is the number of nodes in the tree T'). This asymptotic work bound doesn't help to differentiate the algorithms, or to explain why one of them may be more attractive than Algorithm 0.

Instead, Table 4.3 shows theoretical performance counting the number of node visitations and the number of skip list insertions. The node visitations may be *labelling* (i.e., via **Apply-Consecutive-Labels** or **Apply-Need-Based-Labels**) or the visitation may be **non-labelling** (i.e., via **Compute-Needs**). The node visitations, regardless of purpose, have approximately the same cost, $\mathcal{V}$. Skip list insertions have a higher cost, $\mathcal{I}$.

Table 4.3 also show empirical results from the PROTEUS-to-C transformation engine. The transformations have been divided up into the PROTEUS-to-PROTEUS flattening transformations and the flattened-PROTEUS-to-C transformations. These two sets of transformations were designed to perform very different tasks, so measuring each separately, even though they are part of the same application, will provide an approximation of two completely different sets of transformations.

We see from these data that Algorithm 4 produces a speedup between 2 and 15 times, when compared with Algorithm 0. For this example, the label ranges were large enough so that a total relabelling of the tree was never necessary (so $p(\overline{T'}, \overline{\sigma}) = \overline{\sigma}$). The major difference between the transformation sets was the height of the AST: the height for the second transformation set was approximately 5 times that of the first transformation set, so $h(\overline{T'})$ contributed to the performance difference. As shown in Table 4.1, tree traversal is responsible for approximately 50% of the run-time of the translator, so an infinite speedup in tree traversal would only half the run time of the translator. However, this is a substantial improvement that will be more important when other parts of the translation process are improved.

4.6 Future Work

The current KHEPERA system implements the debugging capabilities described in this dissertation. The system contains more than 20,000 lines of C, `flex`, and `bison` code, implementing about 500 library calls. KHEPERA has been used to implement

Algorithm	Analytic Performance	Average Speedup	
		Transform Set 1	Transform Set 2
0		1.0	1.0
1	$\overline{\sigma}\,\mathcal{I} + \overline{T'}\,\mathcal{V}$	2.6	0.9
2	$\overline{\sigma}\,\mathcal{I} + 2\overline{T'}\,\mathcal{V}$	1.5	0.5
3	$\overline{\sigma}\,\mathcal{I} + (\overline{\sigma} + h(\overline{T'}) + \overline{T'})\,\mathcal{V}$	2.1	0.7
4	$\overline{\sigma}\,\mathcal{I} + (\overline{\sigma} + h(\overline{T'}) + p(\overline{T'}, \overline{\sigma}))\,\mathcal{V}$	15.0	2.3

$\mathcal{V}$ = cost of a node *visitation*

$\mathcal{I}$ = cost of *inserting* into a skip list data structure

$h(\overline{T'})$ is a function of the height of T' at the insertion point for σ, bounded by $\overline{T'}$.

$p(\overline{T'}, \overline{\sigma})$ is a function that returns either $\overline{T'}$ or $\overline{\sigma}$, depending on the need for a relabeling of the whole tree.

Table 4.3: Performance of Fast Tree-Traversal Algorithms

a language processor for a subset of the PROTEUS programming language. In the future, enhancements to various parts of the KHEPERA system could be made:

- The KHEPERA language could be extended to provide more complicated tree-pattern matching capabilities. For example, allowing regular expressions in the tree-pattern specification would allow some types of transformation rules to be more easily specified.

- Currently, the KHEPERA language processor is only partially bootstrapping. An initial version of the processor is built with all of the non-transformational capabilities of the KHEPERA processor. This intermediate processor is used to build the full version of the KHEPERA processor. Other priorities prevented implementation of an additional bootstrapping stage. A fully bootstrapping version would be able to make use of more features in the KHEPERA transformation language to implement more expressive language extensions. One area that was not addressed in the original implementation is the need to insert subtrees into the main AST by appending to other nodes (rather than by simple replacement, as is now supported). Further, type checking could be used to detect several errors which are now only detected when the KHEPERA output is compiled with a C compiler.

- The node-definition file and pretty-printing descriptions that Khepera now uses could be re-designed in light of the experiences gleaned from working with the Proteus-to-C translator. Many transformational programming systems provide some sort of "unparsing" or pretty-printing support, and most of this support appears to be provided in an ad hoc manner. Future research could build on the work of Oppen [1980] and others to provide a more powerful pretty-printing paradigm that would be even more useful for transformational programming.

- In Section 4.5.3.6, several algorithmic improvements were suggested for the final algorithm for rapid tree searching. Additional performance improvements could be obtained by profiling and optimization of low-level library routines. For example, the skip list implementation in Khepera currently does not use "fingers", pointers to recently accessed data that can generally improve the performance of skip lists [Pugh 1990a].

4.7 Conclusion

This chapter introduced the Khepera transformation system, which implements the tracking algorithms from Chapter 3. This implementation demonstrates the practicality and viability of using these algorithms in a real transformational programming tool. Khepera has been used by the author to implement a Proteus-to-C translator, and is now being used by others to implement a Nesl-to-Fortran 90 translator.

One of the problems with using the transformational approach to programming is that a large amount of execution time is spent searching for places to apply the transformations. This problem is discussed in Section 4.5, and an algorithm is presented which allows rapid tree traversal to be performed while still maintaining the same search order as would a standard preorder traversal of the complete tree.

Chapter 5

Debugging with Tracking

In Chapter 3, the notion of translation via program transformation was formalized, and detailed algorithms were presented that track nodes between ASTs. In Chapter 4, the KHEPERA transformation system was presented as an implementation of the low-level tracking algorithms along with several examples of how these algorithms interact with a viewer for the transformation system. In this chapter, the tracking engine is considered as a server that tracks information during translation and that can later answer questions about how nodes map from one tree to another. Applications, such as the KHEPERA viewer, are clients of this server, using the node tracking information provided by the tracking engine in a variety of different ways. For example, using this client-server analogy, the KHEPERA viewer is a client that:

1. uses the node tracking information to map nodes from the original source code to nodes in the transformed output, enabling the setting of a breakpoint;

2. uses the node tracking information to map nodes from the transformed output to the original source code, enabling the determination of current execution location; and

3. uses the node tracking information to navigate between trees, based on how a selected set of nodes changes through the translation process.

The use and implementation of clients that solve two debugging problems will be explored in the next two sections of this chapter:

1. The viewer will be used to demonstrate how node tracking can be used to debug the *program translator* itself. This is a capability of KHEPERA that is not found in modern transformation or debugging systems.

2. Several approaches to the problem of reporting variable values will be described, with a revisitation of the expected behavior data-value problem, and an exploration of loop interchange—an important optimization that is not handled by currently-available expected behavior debugging systems.

5.1 Debugging the Program Translator

The problem of transformation ordering (also called the *phase problem*) is a difficult problem to debug in a transformation system. This problem occurs when the application order of a set of transformations is such that later transformations cannot be applied because earlier transformations removed the opportunity for application. For example, say there are two transformation rules, τ_1 and τ_2. There may be a situation where τ_1 can be applied twice in succession, but that the second application prevents τ_2 from matching. However, if the ordering is changed slightly, so that τ_2 is applied immediately after any application of τ_1, then the rules can all apply in the following order: τ_1, τ_2, τ_1.

The implementation of KHEPERA has been driven by the problem of translating the PROTEUS language into C or FORTRAN 90. PROTEUS is a high-level, nested-data parallel language described by Palmer et al. [1995b,a] and Palmer [1996] that depends on transformations for the flattening of nested-data parallelism. Experience with these transformations was gained by implementing a PROTEUS-to-C translator once using SORCERER [Parr 1997] and again using KHEPERA. In his dissertation, Palmer [1996] identifies "ubiquitous" examples of expressions in the PROTEUS language that are affected by transformation application ordering. Instead of explaining the PROTEUS language here, we will use the example language from Chapter 4 with an additional transformation rule. For brevity, please refer to Section 4.4 (page 68) for the complete description of the language.

The three transformation rules from Section 4.4.3 (page 71) are repeated below, with the addition of a fourth rule:

Rule 1

$$\texttt{(/}\ x_1\ \texttt{in}\ e_1\ \texttt{:}\ x_1\ \texttt{/)} \longrightarrow e_1$$

Rule 2 Provided e_2 is an **Id** or **Int**, and $e_2 \neq x_1$,

$$\texttt{(/}\ x_1\ \texttt{in}\ e_1\ \texttt{:}\ e_2\ \texttt{/)} \quad\longrightarrow\quad \texttt{dist(depth=}\ 0, e_2, \texttt{length(depth=}\ 0, e_1\texttt{))}$$

Rule 3 Provided p is a primitive operation (add, length, range, or dist),

$$\begin{array}{l}\texttt{(/}\ x_1\ \texttt{in}\ e_0\ \texttt{:}\\ \qquad\qquad p\texttt{(}\ \texttt{depth=}\ d,\\ \qquad\qquad\quad e_1,\ldots,e_n\ \texttt{)}\ \texttt{/)}\end{array} \longrightarrow \begin{array}{l}p\texttt{(}\ \texttt{depth=}\ d+1,\\ \quad \texttt{(/}\ x_1\ \texttt{in}\ e_0\ \texttt{:}\ e_1\ \texttt{/)},\\ \quad \ldots,\\ \quad \texttt{(/}\ x_1\ \texttt{in}\ e_0\ \texttt{:}\ e_n\ \texttt{/)}\ \texttt{)}\end{array}$$

Rule 4 Provided e is a complex expression (e.g., not a simple identifier or constant) in which there are no free occurrences of v,

$$\texttt{(/}\ v\ \texttt{in}\ D\ \texttt{:}\ e\ \texttt{/)} \longrightarrow \texttt{dist(}\ \texttt{depth=}\ 0, e, \texttt{length(}\ \texttt{depth=}\ 0, D\texttt{))}$$

For this example, Rule 4 is very similar to Rule 2. In a more complete implementation, Rule 4 would actually introduce another scope into the result, but this is not necessary for this example. For details, see Palmer [1996].

Now, consider the following example, where k is a scalar integer constant:

$$R = \texttt{(/}\ v\ \texttt{in}\ D\ \texttt{:}\ \texttt{(/}\ w\ \texttt{in}\ E\ \texttt{:}\ k+v\ \texttt{/)}\ \texttt{/)}$$

If the rules are applied in the order written (i.e., Rule 1, then Rule 2, then Rule 3, then Rule 4), the transformed output is (the FORTRAN-90 continuation mark (&) has been removed):

```
R = add(depth=2,
        dist(depth=0,
             dist(depth=0, k, length(depth=0, E)),
             length(depth=0, D)),
        dist(depth=1,
             D,
             dist(depth=0,
                  length(depth=0, E),
                  length(depth=0, D))))
```

However, if the ordering is changed so that Rule 4 is applied first (i.e., Rule 4, then Rule 1, then Rule 2, then Rule 3), the transformed output is:

```
R = dist(depth=1,
         add(depth=1,
             dist(depth=0, k, length(depth=0, D)),
             D),
         dist(depth=0,
              length(depth=0, E),
              length(depth=0, D)))
```

While both of these transforms are semantically correct, and will compute the same result, the later transform is more efficient because it performs fewer of the expensive distribution operations. The KHEPERA viewer can be used to understand how transformation ordering affects the translator output.

If the first case, Rule 3 is the first rule that matches, so it is applied, yielding the following result:

```
R = (/ v in D :
       add(depth=1,
           (/ w in E : k /),
           (/ w in E : v /)) /)
```

However, in the second case, Rule 4 is the first rule that matches, so it is applied, yielding the following result:

```
R = (/ v in D :
       dist(depth=0,
            add(depth=0, k, v),
            length(depth=0, E)) /)
```

This key difference, which happens to show up early in this simple example, was discovered by using the KHEPERA viewer to single step between transformations. Similar problems with transformation sequencing have been discovered in much more complicated PROTEUS programs by using the KHEPERA viewer to step between transformations. In complicated cases, where thousands of transformation steps are involved, a portion of the program can be identified, and then most of the intermediate transformations can be skipped using an implementation of the **Find-Next** algorithm described in Section 3.5.2.2 (page 58). After the key transformation step is identified, the ability to step the viewer forward and backward by a single transformation is used to understand which simplification opportunity was being missed. Changes can then be made to the transformation sequencing specification for the translator.

5.2 Reporting Variable Values

Reporting of variable values is a typical debugger capability. The programmer requests a breakpoint be set, and execution of the program stops at that breakpoint. Then the programmer asks for the value of a variable visible from that breakpoint. If the breakpoint is a line, the variable does not have to be on that line—it can be anywhere in the enclosing scope. This problem is, fundamentally, not one that node tracking seeks to solve: the node tracking engine tracks nodes from one AST to another—it does not have the semantic information available to provide information about variables visible from a specific line. Hence, KHEPERA does not provide direct support for the determination of variable currency that is required for expected behavior debugging.

However, KHEPERA can provide answers to queries that are required for expected behavior debugging algorithms or for truthful explanations of how transformations might have changed variable values.

5.2.1 Currency Determination for Scalar Optimizations

Assuming that the transformations implemented in the KHEPERA system fall into the class of simple scalar optimizations that are supported by Adl-Tabatabai [1996], then Adl-Tabatabai's algorithms can be implemented in a KHEPERA client so that variable currency can be determined. This requires considerable overhead above and beyond the implementation of the transformations, but the result is that capabilities are provided that KHEPERA was not originally designed to provide. However, the overhead is not as great as implementing Adl-Tabatabai's expected behavior debugging from scratch, since the KHEPERA system can provide a great deal of support.

To demonstrate how KHEPERA can support the implementation of additional debugging techniques, assume that a set of transformation rules have been written, and the goal is to add the required support for Adl-Tabatabai's data-flow and currency determination algorithms. Adl-Tabatabai outlines four cases where special tracking of *semantic* information is necessary. Below, each case is examined and the changes necessary to track this additional information are described:

Code insertion. Hoisted or sunk expressions must be marked via additions to the transformation rules which hoist and sink expressions (e.g., code motion and CSE).

Code replacement. Replacement of one variable by another must keep track of the original variable. In this case, KHEPERA already provides this sort of tracking automatically, so no changes are needed to the transformation rules.

Code deletion. When an assignment to a variable is eliminated, an addition to the transformation rules must, under certain circumstances, replace this node with a special marker node. The information carried by this marker node is already tracked automatically by KHEPERA, so the only addition is the marker node itself—not its attributes.

Code duplication. When block B is duplicated, marker nodes inside block B must also be duplicated. KHEPERA will perform this duplication without any changes to the transformation rules.

So, out of four types of semantic information that must be tracked, KHEPERA provides at least half of the tracking without any changes. Further, since the KHEPERA system can log all insertions and deletions, KHEPERA can be used to find all of the other places where changes are required.

5.2.2 Variable Values and Complex Transformations

While KHEPERA can support the implementation of debugging algorithms, such as those described by Adl-Tabatabai, these algorithms are useful for only a restricted set of scalar optimizations. They are not useful for aggressive loop transformations or for parallelizing transformations. In these cases, the implementor of the program translator is still faced with the problem of reporting the value of a variable. There are many solutions to this problem. One solution is to simply report the variable's value, without any additional comment. However, if the value of the variable is not current at that point in execution, then the debugger has not provided truthful behavior, since no explanation was given. A simple algorithm based on computing the flow graph of the program is presented below. A program translator will likely have computed a flow graph for other reasons, and that same flow graph can be used to support variable value reporting.

When a breakpoint is reached and the value of a variable x is requested by the user, the debugger interface uses the flow graph to determine the definitions for x that reach the breakpoint (e.g., by following def-use chains). These definitions will be associated with nodes on the AST (e.g., *assignment* nodes that assign a value to x), and these nodes can be tracked from the original tree, T_0, to the final transformed tree, T_ℓ. The user can then be shown the definitions on T_0 and what those definitions track to on T_ℓ. This will help the user to understand the value for x reported by the debugger.

As a useful refinement, the definitions for x in T_ℓ can also be determined by using flow graph analysis on the transformation output. The definition points on T_ℓ can also be shown to the user and compared with the definition point on T_0. If these definition points differ, then the variable x may not be current at the breakpoint in T_ℓ.

The examples in the next section will help to clarify how this algorithm might be used, how it helps the end-user to interpret the reported value for the variable queried, and how it differs from the information that Adl-Tabatabai's algorithms would provide.

5.2.3 Solving Data-Value Problems With KHEPERA

In general, data-value problems are caused by assignments that are either deleted (e.g., via redundant or dead assignment elimination) or moved (e.g., via code hoisting) by the transformations. Examples of these sorts of problems were presented in Section 2.3.3 (page 15), with a description of typical expected behavior solutions. In this section, solutions to these examples using the truthful behavior algorithm based on simple flow graph analysis from Section 5.2.2 will be discussed.

Additionally, examples of variable promotion (e.g., from **scalar** to **vector**) and loop interchange will be explored, since these transformations are not covered by expected behavior research, and therefore demonstrate some of the additional capabilities provided by KHEPERA.

```
100 ...                          →   100 ...
110 x = y + z;                       110 x = y + z;
120 ...                              120 ...
130 x = y + z;
140 ...                              140 ...
```

Figure 5.1: Redundant Assignment Elimination

5.2.3.1 Redundant Assignment Elimination

The example of redundant assignment elimination from Section 2.3.4.1 (page 18) is shown in Figure 5.1. Here, line 130 is removed because x, y, and z were not modified since the assignment in line 110.

Assuming a breakpoint on line 140, the truthful behavior algorithm can examine the assignment to x on lines 110 and 130. The x on line 110 would be tracked to the x on line 110, showing the user that the assignment was not transformed. The x on line 130 could be tracked to two different places in the transformed code, depending on how the transformation rule was written:

- If the transformation rule matches the assignment to x on line 110, and then removes all redundant assignments, then the assignment to x on line 130 would track to the assignment on line 110 (because that is where the rule matched).

- If the transformation rule matches the assignment to x on line 130 and then deletes the redundant assignment, tracking would move to a point higher in the tree than the deleted line, tracking to line 120.

In both cases, an examination of the definition of x during the breakpoint in line 140 would show that line 130 was removed, and would provide some explanation for the end-user. For the naïve user, this explanation could take the simple form of reporting that the assignment on line 130 was removed because the "redundant assignment removal" transformation had been applied. For the sophisticated user, or for the program transformation implementor, a "before and after" view of the local portion of the program could be displayed, showing the deleted assignment.

```
200 ...                  ⟶  200 ...
210 x = w - v;
220 ...                      220 ...
230 x = y + z;               230 x = y + z;
240 ...                      240 ...
```

Figure 5.2: Dead Assignment Elimination

5.2.3.2 Dead Assignment Elimination

The example of dead assignment elimination from Section 2.3.4.2 (page 18) is shown in Figure 5.2, where x is not used between line 210 and 230.

Without any such semantic knowledge, the truthful behavior algorithm can use the tracking system to analyze the transformation of x on line 210. Again, there are two possible ways to write the transformation rule:

- If the transformation rule matches line 230 and then eliminates line 210, x on line 210 will track to x on line 230. A report can be made to the user, either with a simple explanation about the application of the "dead store elimination" rule, or with a more complex before and after view of the transformation process.

- If the transformation rule matches the x on line 210 and then eliminates that line, tracking for x will move to line 200, since that is the "parent" of the eliminated line on the AST (see Figure 3.9 on page 53). In this case, the movement of the tracking point and the deletion of the line can be reported to the user.

In both cases, tracking can be combined with data flow analysis to provide a reasonable explanation for the user without any special debugging considerations when the transformation rule was written.

5.2.3.3 Code Hoisting

The example of code hoisting from Section 2.3.4.3 (page 19) is shown in Figure 5.3, where x is not used during the **else** part.

There are two interesting breakpoints in this example:

Line 360. The expected value of x is $u - v$, but the actual value is $y + z$.

```
300 x = u - v;             -->   300 x = u - v
310 if (c) {                     310 if (c) {
320    x = y + z;                320    x = y + z
330 } else {                     330 } else {
340    ...                       340    ...
                                 350    x = y + z;
360    ...                       360    ...
370 }                            370 }
380 ...                          380 ...
390 x = y + z;
400 ...                          400 ...
```

Figure 5.3: Code Hoisting

Line 380. The expected value of x is *either* $u - v$ or $y + z$, depending on which branch of the **if** was taken. The actual value is $y + z$. This may lead the programmer to believe that the first branch was always taken, when, in fact, the transformations make this assumption incorrect.

Using the truthful behavior algorithm, data flow analysis can be used to show that there are two definitions for x in the original source (on lines 300 and 320). These definitions will be tracked to the corresponding definitions in the final transformed program (also on lines 300 and 320). Further, data flow analysis of the final transformed program will show that there are *three* definitions for x visible from line 360 or line 380. This additional definition can be tracked backward to T_0, allowing the debugger to display before and after views that show all of the important assignments to x, even those that come after line 380 in the original program.

5.2.3.4 Variable Promotion

An example of variable promotion performed by a loop vectorization transformation is shown in Figure 5.4. T is a scalar in the original program, but T' is a vector.

Providing expected behavior debugging for this type of transformation is difficult. After setting a breakpoint on line 510, the expectation is that the loop will stop during the first iteration. Reconstructing this expected behavior in the general case is difficult, and is not usually discussed in the literature dealing with expected behavior debugging.

A, B, C: **array** (1..n) of **integer**
T: **integer**
do $i = 1, n$
    $T = A(i) + B(i)$
    $C(i) = C(i) + y * T$
end do

$\longrightarrow$

A, B, C, T': **array** (1..n) of **integer**
$T' = A + B$
$C = C + y * T'$

Figure 5.4: Example Loop Vectorization

```
10 do J = 2, M
20     do I = 1, N
30         A(I,J) = A(I,J-1) + B(I,J)
40     enddo
50 enddo
```

$\longrightarrow$

```
10 do I = 1, N
20     do J = 2, M
30         A(I,J) = A(I,J-1) + B(I,J)
40     enddo
50 enddo
```

Figure 5.5: Increasing Parallelism with Loop Interchange

Given a breakpoint at line 510 and a request for the value of variable i, the truthful behavior algorithm would report that the **do** loop was replaced. A request for the value of variable T would report that the defining assignment to T was changed to a defining assignment to variable T', thereby providing the debugger enough information to explain that a variable substitution occurred. The debugger, with knowledge of the type system, could then offer to display some part of T'.

5.2.3.5 Loop Interchange

Wolfe [1996] calls loop interchange the "single most important loop restructuring transformation". Traditionally, loop interchange has been important for the discovery of parallelism: if the inner loop carried dependencies, but the outer loop did not, then switching the loops would allow the inner loop to execute in parallel. An example of this use of loop interchange is shown in Figure 5.5. Before the loop interchange, the inner loops cannot be executed in parallel because $A(I, J-1)$ must be computed before $A(I, J)$. Hence, the iteration over J must proceed sequentially. After the loop interchange, however, all of the inner loops can execute in parallel.

Loop interchange is also important for scalar compilers. As branch prediction in processors becomes more sophisticated, interchange to replace an inner loop that iterates only a few times with an outer loop that iterates many times can dramatically increase performance [Intel Corporation 1997]. Further, as processor speeds continue to increase faster than memory speeds [Wulf and McKee 1994; McCalpin 1995], loop interchange can be used to increase the spatial and sequential locality of memory references (e.g., by reducing the stride of the loop to one [Bacon et al. 1994]).

A Khepera transformation rule that performs the loop interchange shown in Figure 5.5 might match the outer loop, perform some analysis of the contents of the loop (e.g., verify that loop interchange is helpful and allowed, and verify, for this simple example, that `id1` is not used within the expressions that define the inner loop), and then perform a replacement. A rule that matches a simple case where loop interchange might be useful would look something like this:

```
match (outer:N_For
       id1:N_Identifier
       lower1:N_Expression
       upper1:N_Expression
       (N_StatementBlock
        (inner:N_For
         id2:N_Identifier
         lower2:N_Expression
         upper2:N_Expression
         body:.)
         rest:.))
when (is_interchange_ok(outer))
build new with (N_For id2 lower2 upper2
                (N_StatementBlock
                 (N_For id1 lower1 upper1 body)
                 rest))
replace outer with new
```

In this case, asking questions during debugging about I and J would track to the appropriate variables in the interchanged loops. Asking questions about either **for** statement itself would track to the **for** in the outer loop (since the `N_For` nodes were not copied in the rule).

In this simple example, Khepera would provide a simple, easy-to-understand answer to a debugging query. An expected behavior debugging system, however, would probably not support loop interchange transformations at all because reconstructing the behavior of the non-interchanged loops, especially in the face of many composed transformations, would be difficult or impossible. Adl-Tabatabai leaves to future work the determination of currency in the face of loop-nest transformations, including loop interchange and loop skewing. If algorithms for currency determination in the face of loop transformations were available, the methods would probably involve transformation-specific annotations that track *semantic* information about the transformation itself. In contrast, the answers provided by KHEPERA are independent of the specific transformation semantics. Again, as other methods are discovered to provide more debugging information to the user, these methods can be added to the KHEPERA rules to provide more information (although at the cost of increased overhead for the transformation implementor and maintainer).

5.3 Conclusion

In addition to the ability to answer questions about simple scalar optimizations using information that is tracked transparently, the node tracking of the KHEPERA system also has the ability to provide debugging features that other systems cannot provide:

- KHEPERA can answer debugging questions about the *transformation system* itself, thereby providing support for debugging the program translator and for gaining an understanding of the transformations. This support can be used by the implementor of the program transformation system or by the sophisticated end-user who is interested in understanding the workings of the program translator.

- KHEPERA can provide valuable scaffolding for the implementation of debugging systems that require *semantic* information about the transformations being applied.

- Without the addition of *semantic* tracking to the individual transformations, KHEPERA can be combined with data flow analysis to provide truthful information about variable values.

- KHEPERA can answer debugging questions for optimizations and transformations that are more complex than the simple scalar optimizations discussed in the literature. Examples include loop interchange, variable promotion, and the flattening of nested-data parallelism.

Chapter 6

Contributions and Future Work

Translators are pervasive, being used to implement increasingly complicated languages and language extensions. I have focused on truthful behavior debugging, since this type of behavior is reasonable for debugging a program undergoing large structure changes. This is also the kind of debugging required by a translator implementor.

Assuming the translator is implemented as a series of tree transformations, my algorithms track debugging information at a very low level, without semantic knowledge of the languages being transformed. This saves the implementor from the task of writing, for each and every transformation, additional code to implement debugging. It also provides a framework for implementing more complicated semantic-aware debugging systems if the implementor chooses.

I have built KHEPERA as an example implementation, showing that this automatic and transparent debugging is possible within a real transformation system.

6.1 Contributions

The work presented in this dissertation makes the following contributions:

1. Given a program translator that is structured as a set of program transformations operating on a tree-based representation, I have described methods for *tracking* debugging information in this system in a manner that is transparent, automatic, and independent of the semantics of the languages being transformed (see Chapter 3).

2. I have presented algorithms that use this *tracking* information to provide support for debugging the translator and the output of the translator (see Chapter 4).

3. I have shown how to use this *tracking* framework to systematically build advanced debugging support, relying on the semantics of the languages and transformations (see Chapter 5).

4. I have described algorithms that increase the performance of the transformation process (see Chapter 4).

6.2 Evaluation

To evaluate this approach and the proposed algorithms:

1. I have constructed KHEPERA (see Chapter 4), a program transformation system with integral support for the construction of debuggers. This system implements the *tracking* algorithms (see Chapter 3), and the algorithms for rapid tree walking (see Section 4.5.3). KHEPERA has been used to implement a PROTEUS-to-C translator.

2. I have written a viewer for the KHEPERA system that can be used to debug the translator implementation (see Chapter 5).

3. I have used the KHEPERA system and its viewer to explore debugging capabilities for traditional compiler optimizations, for more aggressive loop and parallelizing transformations, and for the transformation process itself (see Chapter 5).

4. I have used the KHEPERA system to analyze average performance for the rapid tree-walking algorithms for a set of programs and transformations (see Section 4.5.3).

6.3 Future Work

6.3.1 KHEPERA Improvements

I've built KHEPERA as an initial demonstration of the tracking algorithms. KHEPERA can be improved by making the language for describing transformations more expressive, by adding a better way to describe how to sequence transformations, and by more work with data structures to improve its overall performance.

Further, the algorithms described in Chapter 4 for rapid tree-walking can be improved with more careful implementation of the underlying data structures. For example, skip list performance can be dramatically improved with the addition of "fingers" that cache recently accessed locations in the skip list structure [Pugh 1990a].

6.3.2 Tracking Improvements

The tracking algorithms presented here should also be applicable to translators that generate machine code. Experiments with these algorithms and machine code generation can take two routes:

1. the exploration of tracking machine code on an AST, and
2. the exploration of using the tracking algorithms on an IR that is not an AST (often, non-AST IRs are used for the final stages of machine code generation and optimization).

More experience should be obtained with debuggers for large systems, and for systems other than the PROTEUS-to-C translator. The tracking algorithms should be incorporated into another translator construction system, such as the SUIF compiler construction toolkit [Tjiang et al. 1992].

6.3.3 Program Verification

During transformation, KHEPERA produces a log of transformations applied to various parts of the tree. This log can be "replayed" to transform the original program into the transformed program. Discovering which transforms to apply to the program may be a difficult and complicated process, involving type exploration and program analysis. However, the log of transformations is a much simpler problem. Proving the correctness of a small "transformation application" program would be much easier than proving the correctness of the KHEPERA system. To prove correctness of a program P translated to P' by KHEPERA—that is, to prove that P and P' are semantically equivalent—we need only prove:

1. the correctness of the transformation rules, and
2. the correctness of the program applying the log of transformations.

Appendix A

The Khepera Language

The Khepera language is described in this section. Section 4.4 (page 68) discusses how some of the Khepera language constructs can be used to build a language processor.

A.1 Reserved Words

The following words are reserved outside of C code sections:

`break`	`build`	`children`	`decl`
`if`	`include`	`match`	`rebuild`
`replace`	`return`	`rule`	`using`
`walk`	`when`		

A.2 Reserved Variables

Variable names beginning with `_kh_` are reserved everywhere, including within the C code sections.

A.3 Comments

Standard C comments (e.g., `/* */`) may be used anywhere.

A.4 Tree-Matching Specifications

$$
\begin{array}{lll}
\textit{tree} ::= \texttt{(}\ \textit{node}\ [\ \textit{children}\ ]\ \texttt{)} & & \\
\textit{children} ::= [\ \textit{children}\ ]\ \textit{node} & & \\
\qquad\qquad \mid\ \ [\ \textit{children}\ ]\ \textit{tree} & & \\
\textit{node} ::= \textit{id} & & \text{[AST node name]} \\
\qquad \mid & \textit{id}\ :\ \textit{id} & \text{[Label preceding AST node name]} \\
\qquad \mid & . & \text{[Wildcard (matches any single node)]} \\
\qquad \mid & \textit{id}\ :\ . & \text{[Label preceding wildcard]} \\
\qquad \mid & .. & \text{[Sibling wildcard (matches all remaining siblings)]} \\
\qquad \mid & \textit{id}\ :\ .. & \text{[Label preceding sibling wildcard]} \\
\qquad \mid & \texttt{0} & \text{[Ground (matches absence of node)]}
\end{array}
$$

When matching an AST, the first node in a *tree* must be a node name from the node definition file. Our local convention is that all of these special names start with `N_` to distinguish them from other identifiers, but this convention is not enforced by KHEPERA.

One should think of a sibling wildcard as a reference to a list or forest of subtrees.

A.5 Tree-Building Specifications

$$
\begin{array}{lll}
\textit{b-tree} ::= \texttt{(}\ \textit{b-head}\ [\ \textit{b-children}\ ]\ \texttt{)} & & \\
\textit{b-children} ::= [\ \textit{b-children}\ ]\ \textit{b-node} & & \\
\qquad\qquad \mid\ \ [\ \textit{b-children}\ ]\ \textit{b-tree} & & \\
\textit{b-head} ::= \textit{id} & & \text{[AST node reference]} \\
\qquad \mid & \textit{id}\ :\ \textit{id} & \text{[New label preceding AST node name]} \\
\textit{b-node} ::= \textit{id} & & \text{[Matched label]} \\
\qquad \mid & \textit{id}\ :\ \textit{id} & \text{[New label preceding matched label]}
\end{array}
$$

Labels from the tree-matching specification may be used in this section to indicate that a *copy* of the matched tree should be included at this point in the tree that is being built. A "new label" may be attached to that *copy* so that the *copy* may be referenced in later C code. If a label from the tree-matching specification refers to a sibling wildcard, then the labeled tree and all rightward siblings will be copied.

A.6 Rule Specifications

rule ::= `rule` *id* `{` *command-list* `}`
match ::= `match` *tree* [`when` *c-bool*]
command-list ::= [*command-list*] *command*

command ::=	`decl` `{` *c-code* `}`	[C declarations]
|	*match*	[Match a subtree]
|	`build` *id* `with` *b-tree*	[Build a new subtree]
|	`rebuild` *id* `with` *b-tree*	[Rebuild a subtree]
|	`using` *id* *match* *command*	[Using a matched label, perform another match]
|	`children` *id* *command*	[Iterate over children]
|	`break`	[Break from iteration]
|	`do` `{` *c-code* `}`	[Arbitrary C]
|	`replace` [id_1] `with` id_2	[Replace tree]
|	`{` *command-list* `}`	
|	`if` *c-bool* *command*	[Conditional]
|	`return`	[Early return]

If id_1 is missing from a `replace` command, then the subtree matched by the first `match` command is used.

c-code can be any arbitrary C code. At this time, this code is not parsed by KHEPERA, so any errors in this code will be reported at compile-time.

c-bool can be an arbitrary C expression. This expression will be used in a C `if` statement to guard the `match`. Since KHEPERA rules are generally executed in an iterative manner until they no longer "fire", it is extremely important that the `when` clause prevents the rule from firing when there is not more work to be done—otherwise the iterative applications will halt only if some *other* rule removes the pattern which is matched by the current rule.

The C code can refer to labels used in *tree* and *b-tree* constructs. These labels are seen a KHEPERA "Node" variables (e.g., a pointer to a node created with `tre_mk`).

The `using` command begins a new scope, so variable names may be re-used from one `using` command to another. This often makes the KHEPERA rules easier to read.

Appendix B

Optimizations

B.1 Common Scalar Optimizations

This section provides brief descriptions of representative scalar optimizations that are commonly used in production compilers [Stallman 1993]. Methods for providing expected behavior debugging in the face of these and similar optimizations have been described [Zellweger 1984; Copperman 1993a; Adl-Tabatabai 1996].

B.1.1 Constant Folding

Constant folding (also called *constant-expression evaluation*) is the compile-time evaluation of expressions whose values are known to be constant [Muchnick 1997, p. 329].

B.1.2 Copy Propagation

Copy propagation (also called *assignment propagation*) replaces the use of a variable with the expression most recently assigned to it [Muchnick 1997, p. 356]. For example, given the assignment $x = y$, copy propagation would replace later uses of x with y.

B.1.3 Constant Propagation

Constant propagation replaces the use of a variable with the constant value most recently assigned to it [Muchnick 1997, p. 362].

B.1.4 Common-Subexpression Elimination

Common-subexpression elimination locates multiple occurrences of the same expression (common-subexpression), and replaces the recomputation with the use of a value stored from the initial computation [Muchnick 1997, p. 378].

B.1.5 Dead Assignment Elimination

Dead assignment elimination locates assignments to variables which are never used on any path from assignment, and removes the assignment [Muchnick 1997, p. 592]. An example is shown in Figure 2.8 (page 18).

B.1.6 Dead Code Elimination

Dead code elimination locates code which is never executed or which computes values which are never used on any path from the code, and removes the code [Muchnick 1997, p. 592].

B.1.7 Procedure Inlining

Procedure inlining (also called *procedure integration* or *automatic inlining*) replaces a call to a procedure with a copy of the procedure body [Muchnick 1997, p. 465].

B.1.8 Cross-Jumping

"Cross-jumping is a special case of procedure discovery that examines code paths that join. If the tail portions of any two paths are the same, cross-jumping moves the join point for those two paths from its original location backward to the earliest identical point and deletes one copy of the identical code". [Zellweger 1984, pp. 54–5]. Cross-jumping is also called *tail merging* [Muchnick 1997, p. 590]. An example of cross-jumping is shown in Figure 2.4 (page 14).

B.1.9 Strength Reduction

Strength reduction replaces one expression with another expression that is equivalent but uses a less expensive operator [Bacon et al. 1994, p. 359].

B.1.10 Induction Variable Elimination

An *induction variable* is a variable whose value is derived from the number of iterations that have been executed by an enclosing loop. After strength reduction has been performed on induction variable expressions, the induction variables can often be eliminated entirely. In this case, loop termination relies on a strength-reduced expression instead of on the original induction variable [Bacon et al. 1994, p. 359].

B.1.11 Loop-Invariant Code Motion

When an expression is computed within a loop, but the value computed does not change between loops, the expression can be moved outside the loop [Bacon et al. 1994, p. 360].

B.1.12 Code Hoisting

Code hoisting (also called *unification*) finds expression that would always be evaluated on some path through a program, and moves the expressions to the earliest possible point beyond which they would always be evaluated [Muchnick 1997, p. 417]. An example is shown in Figure 2.9 (page 19).

B.1.13 Loop Unswitching

Loop unswitching replaces a loop which contains a loop-invariant conditional with a conditional containing a copy of the loop in each of its branches [Bacon et al. 1994, p. 361].

B.1.14 Loop Unrolling

Loop unrolling replicates the body of a loop some number of times, with a corresponding change in the loop bounds and the use of the index variables [Bacon et al. 1994, pp. 368–9].

B.1.15 Loop Peeling

Loop peeling removes a small number of iterations from the beginning or end of a loop, replicating the code before or after the main body of the loop [Bacon et al. 1994, p. 372].

B.2 Aggressive Loop Optimizations

This section describes aggressive loop optimizations for which no expected behavior debugging methods exist [Zellweger 1984; Copperman 1993a; Adl-Tabatabai 1996].

B.2.1 Loop Interchange

Loop interchange exchanges the position of two loops. An example is shown in Figure 5.5 (page 127).

B.2.2 Loop Skewing

Loop skewing changes the bounds of the loop together with the expressions that use the corresponding index variables [Wolfe 1996, pp. 341–3]. Loop skewing is often used an an "enabling" transformation that is useful in combination with loop interchange [Bacon et al. 1994, pp. 363–4].

B.2.3 Loop Reversal

Loop reversal changes the direction in which loops iterate over their index variable, and can be used to change the dependencies for vectors within a loop, thereby enabling other optimizations [Bacon et al. 1994, p.365].

B.2.4 Loop Coalescing

Loop coalescing (also called *loop collapsing*) replaces a pair of nested loops with a single loop [Bacon et al. 1994, p. 371].

B.2.5 Strip Mining

Strip mining replaces a single loop with two nested loops [Wolfe 1996, pp. 350–1].

B.2.6 Loop Tiling

Loop tiling (also called *loop blocking*) is similar to strip mining, but it operates on multiple nested loops instead of a single loop. Loop tiling can be an important optimization on scalar machines, since it can improve cache reuse [Bacon et al. 1994, pp. 366–7].

B.2.7 Loop Splitting

Loop splitting (also called *loop fission* or *loop distribution*) replaces a single loop with several (non-nested) loops: each loop has the same range as the original loop, but contains only a subset of the statements which were in the original loop [Bacon et al. 1994, p. 367].

B.2.8 Loop Jamming

Loop jamming (also called *loop fusion*) is the inverse of loop splitting: several loops with the same bounds are replaced with a single loop containing the union of the statements in the original loop.

B.2.9 Software Pipelining

In software pipelining, the body of a loop is broken up into stages and the original loop is replaced by a new loop that intermingles the stages across the iteration space [Bacon et al. 1994, p. 369].

B.3 Transformations for Flattening Nested-Data Parallelism

This section describes aggressive parallelizing optimizations for which no expected behavior debugging methods exist [Zellweger 1984; Copperman 1993a; Adl-Tabatabai 1996].

Flattening nested-data parallelism transforms a nested parallel construct into a flat parallel construct [Blelloch 1990, p. 143]. Blelloch and Sabot [1990] first introduced the concept of flattening nested-data parallelism. Prins and Palmer [1993] and Palmer [1996] present flattening in terms of a transformational framework with two distinct sets of transformations that are necessary to flatten a program: the first is the elimination of *apply-to-all* constructs (also called *iterator elimination*); the second is the *promotion of functions* (also called *replication*).

B.3.1 Apply-to-All Elimination

Apply-to-all constructs are replaced with calls to special functions which operate on a set of inputs in parallel [Palmer 1996, Chapter 3]. For example, the apply-to-all construct:

$$\texttt{(/}\ i\ \texttt{in}\ D\ \texttt{:}\ i+i\ \texttt{/)}$$

can be replaced by a call to a special `plus` function which can operate on all elements in a sequence in parallel:

$$\texttt{plus}(D, D)$$

B.3.2 Promotion of Functions

The flattening of nested-data parallelism depends on the existence of versions of routines which can operate on a set of inputs in parallel. Blelloch [1990] calls the creation of these routines "replication". The creation of these special functions is called *promotion* or *introduction of data-parallel function definitions*. For example, if the following apply-to-all construct was eliminated:

$$\texttt{(/}\ i\ \texttt{in}\ D\ \texttt{:}\ f(i,i)\ \texttt{/)}$$

to yield:

$$f'(D, D)$$

then the promotion transformation would copy the source code for f and create the special version, f'.

Appendix C

Obtaining the KHEPERA Transformation System

The programs in the KHEPERA Transformation System are licensed under the terms of the GNU General Public License and the library routines in KHEPERA and LIBMAA are licensed under the GNU Library General Public License. Copies of these licenses are included in the source code distributions.
The KHEPERA system is currently available for anonymous ftp from:

`ftp://ftp.cs.unc.edu/pub/projects/khepera`

Web pages with pointers to the KHEPERA source distribution include:

`http://www.cs.unc.edu/~faith/khepera`
`http://www.cs.unc.edu/Research/khepera`

If these pointers have changed over time, please send email to Jan F. Prins at `prins@cs.unc.edu` or to Rickard E. Faith at `faith@cs.unc.edu` or `faith@acm.org`.

Appendix D

Mythology

Kheperȧ, 𓆣𓂋𓇋𓀭, is the third form of Rā, the Sun-god, and is called the "father of the gods". Kheperȧ rose up out of the watery abyss of Nu and created Maā, 𓆄𓀭, as a foundation upon which to create everything else [Budge 1969, pp. 295–8]. Kheperȧ is self-begotten and self-born, and is associated with creation, rebirth, and transformation. His symbol is the Egyptian Scarab beetle (*Scarabaeus sacer*), a symbol of creation and transformation [Budge 1969, pp. 355–8].

Bibliography

Adams, J. C., Brainerd, W. S., Martin, J. T., Smith, B. T., and Wagener, J. L. 1992. *Fortran 90 handbook: complete ANSI/ISO reference*. Intertext Publications/McGraw-Hill Book Company, New York.

Adl-Tabatabai, A.-R. 1996. *Source-level debugging of globally optimized code*. Ph.D. dissertation, published as Technical report CMU-CS-96-133 (20 June). School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Adl-Tabatabai, A.-R. and Gross, T. 1994. Symbolic debugging of globally optimized code: data value problems and their solutions. Technical report CMU-CS-94-105 (January). School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Appel, A. W. 1997. *Modern compiler implementation in Java: basic techniques*. Cambridge University Press, Cambridge, United Kingdom. Preliminary edition.

Bacon, D. F., Graham, S. L., and Sharp, O. J. 1994. Compiler transformations for high-performance computing. *ACM Computing Surveys 26*, 4 (December), 345–420.

Balzer, R. M. 1969. EXDAMS—EXtendable Debugging and Monitoring System. *Proceedings of the AFIPS Spring Joint Computer Conference* (Boston, Massachusetts, 14–16 May 1969), volume 34, pages 567–80. AFIPS Press.

Bates, R. M. 1996. Examining the Cocktail toolbox: tools for producing compilers, translators, and more. *Dr. Dobb's Journal 21*, 3 (March), 78, 80–2, 95–6.

Bentley, J. 1988. *More programming pearls: confessions of a coder*. Addison-Wesley Publishing Company, Reading, Massachusetts.

Bentley, J. L., Jelinski, L. W., and Kernighan, B. W. 1987. CHEM—a program for phototypesetting chemical structure diagrams. *Computers and Chemistry 11*, 4, 281–97.

Bertot, Y. 1991. Occurrences in debugger specifications. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation* (Toronto, Ontario, Canada, 26–28 June 1991). Published as *SIGPLAN Notices 26*, 6 (June), 327–37.

Blelloch, G. E. 1990. *Vector models for data-parallel computing.* MIT Press, Cambridge, Massachusetts.

Blelloch, G. E. and Sabot, G. W. 1990. Compiling collection-oriented languages onto massively parallel computers. *Journal of Parallel and Distributed Computing 8*, 2 (February), 119–34.

Bozkus, Z., Meadows, L., Nakamoto, S., Schuster, V., and Young, M. 1995. Compiling High Performance Fortran. *Proceedings of the 7th SIAM Conference on Parallel Processing for Scientific Computing* (San Francisco, CA, 15–17 February 1995), pages 704–9. Society for Industrial and Applied Mathematics, Philadelphia.

Brooks, G., Hansen, G. J., and Simmons, S. 1992. A new approach to debugging optimized code. *SIGPLAN '92 Conference on Programming Languages Design and Implementation* (San Francisco, California, 17–19 June 1992). Published as *SIGPLAN Notices 27*, 7 (July), 1–11. This work is also discussed in US Patent 5371747: Debugger program which includes correlation of computer program source code with optimized object code.

Budge, E. A. W. 1969. *The gods of the Egyptians or studies in Egyptian mythology*, volume 1. Dover Publications, New York.

Cameron, R. D. 1988. An abstract pretty printer. *IEEE Software 5*, 6 (November), 61–7.

Cann, D. C. 1992. *The optimizing SISAL compiler: version 12.0.* Lawrence Livermore National Laboratory. This manual is available from `ftp://sisal.llnl.gov/pub/-sisal/MANUAL.12.7.tar.Z`. More information on Sisal is available from `http://-www.llnl.gov/sisal/`.

Cardelli, L. 1987. Basic polymorphic typechecking. *Science of Computer Programming 8*, 2 (April), 147–72. A revised version is available from `http://research.microsoft.com/research/cambridge/luca/Papers/-BasicTypechecking.ps`.

Cohn, R. 1992. *Source-level debugging of automatically parallelized programs.* Ph.D. dissertation, published as Technical report CMU-CS-92-204 (23 October). School of Computer Science, Carnegie Mellon University, Pittsburgh, Pennsylvania.

Cool, L. E. 1992. Debugging VLIW code after instruction scheduling. Technical report 92-TH-009 (July). Oregon Graduate Institute of Science & Technology, Portland. Master's thesis.

Copperman, M. 1993a. *Debugging optimized code without being misled.* Ph.D. dissertation, published as Technical report UCSC-CRL-93-21 (11 June). Board of Studies in Computer and Information Sciences, University of California, Santa Cruz.

Copperman, M. 1993b. Debugging optimized code without being misled: currency determination. Technical report UCSC-CRL-93-24. Baskin Center for Computer Engineering & Information Sciences, University of California, Santa Cruz.

Copperman, M. 1994. Debugging optimized code without being misled. *ACM Transactions on Programming Languages and Systems 16*, 3 (May), 387–427.

Copperman, M. and McDowell, C. E. 1993. A further note on Hennessy's "Symbolic debugging of optimized code". *ACM Transactions on Programming Languages and Systems 15*, 2 (April), 357–65.

Cordy, J. R. and Carmichael, I. H. 1993. *The TXL programming language syntax and informal semantics, version 7* (June). Software Technology Laboratory, Department of Computing and Information Science, Queen's University at Kingston, Ontario. This manual is available from `ftp://ftp.qucis.queensu.ca/-pub/txl/TXL7manual.ps`. More information on TXL is available from `http://-www.qucis.queensu.ca/home/stlab/TXL/`.

Cordy, J. R., Halpern-Hamu, C. D., and Promislow, E. 1991. TXL: a rapid prototyping system for programming language dialects. *Computer Languages 16*, 1 (January), 97–107.

Cormen, T. H., Leiserson, C. E., and Rivest, R. L. 1991. *Introduction to algorithms.* MIT Press, Cambridge, Massachusetts; McGraw-Hill Book Company, New York.

Coutant, D. S., Meloy, S., and Ruscetta, M. 1988. DOC: a practical approach to source-level debugging of globally optimized code. *SIGPLAN '88 Conference on Programming Language Design and Implementation* (Atlanta, Georgia, 22–24 June 1988). Published as *SIGPLAN Notices 23*, 7 (July), 125–34. This work is also discussed in US Patent 4953084: Method and apparatus using variable ranges to support symbolic debugging of optimized code.

Edelstein, O., Gafni, Y., Rainish, V., and Zernik, D. 1992. Source level debugging of optimized code. Unpublished manuscript.

Elmasri, R. and Navathe, S. B. 1989. *Fundamentals of database systems.* Benjamin/Cummings Publishing Company, Redwood City, California.

Faith, R. E. 1996a. Libmaa User's Manual. Technical report TR96-009. Department of Computer Science, University of North Carolina at Chapel Hill.

Faith, R. E. 1996b. The KHEPERA Transformation System. Technical report TR96-010. Department of Computer Science, University of North Carolina at Chapel Hill.

Faith, R. E., Nyland, L. S., and Prins, J. F. 1997. KHEPERA: a system for rapid implementation of domain specific languages. *Conference on Domain-Specific Languages*

(DSL) (Santa Barbara, California, 15–17 October 1997), pages 243–55, Ramming, C., editor. USENIX.

Feiler, P. H. 1982. *A language-oriented interactive programming environment based on compilation technology*. Ph.D. dissertation, published as Technical report CMS-CS-82-117 (May). Carnegie-Mellon University, Pittsburgh, Pennsylvania.

Feldman, S. I., Gay, D. M., Maimone, M. W., and Schryer, N. L. 1995. A Fortran-to-C converter. Technical report 149 (22 March). AT&T Bell Laboratories, Murray Hill, New Jersey. Available from `http://www.netlib.org/f2c/f2c.ps`.

Fritzson, P. 1983. A systematic approach to advanced debugging through incremental compilation. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on High-Level Debugging* (Pacific Grove, CA, 20–23 March 1983). Published as Johnson, M. S., editor, *SIGPLAN Notices 18*, 8 (August), 130–8.

Gannon, D., Yang, S. X., and Beckman, P. 1994. *User guide for a portable parallel C++ programming system, pC++* (9 September). Indiana University, Bloomington, Indiana. This user guide is available from `ftp://-ftp.extreme.indiana.edu/pub/sage/pcxx_ug.ps`. More information is available from `http://www.extreme.indiana.edu/`.

Grosch, J. and Emmelmann, H. 1990. *A tool box for compiler construction* (21 January), Compiler Generation Report No. 20. GMD Forschungsstelle an der Universität Karlsruhe.

Gupta, R. 1988. Debugging code reorganized by a trace scheduling compiler. *3rd International Conference on Supercomputing (Proceedings, Supercomputing '88)* (1988), volume III (Supercomputer Design: Hardware & Software), pages 422–30, Kartashev, L. P. and Kartashev, S. I., editors. International Supercomputing Institute, St. Petersburg, Florida.

Harbison, S. 1990. Modula-3. *Byte 15*, 12 (November), 385–8, 390, 392.

Henderson, F., Somogyi, Z., and Conway, T. 1995. Compiling logic programs to C using GNU C as a portable assembler. *Proceedings of the ILPS '95 Postconference Workshop on Sequential Implementation Technologies for Logic Programming Languages* (Portland, Oregon, December 1995). This paper is available from `http://www.cs.mu.oz.au/research/mercury/papers/mercury_to_c.ps.gz`. For more information on Mercury, see `http://www.cs.mu.oz.au/research/mercury/`.

Hennessy, J. 1982. Symbolic debugging of optimized code. *ACM Transactions on Programming Languages and Systems 4*, 3 (July), 323–44.

Hindley, R. 1969. The principal type-scheme of an object in combinatory logic. *Transactions of the American Mathematical Society 146*, 29–60.

Hölzle, U., Chambers, C., and Ungar, D. 1992. Debugging optimized code with dynamic deoptimization. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation* (San Francisco, California, 17–19 June 1992). Published as *SIGPLAN Notices 27*, 7 (July), 32–43.

Intel Corporation. 1997. *Intel architecture optimization manual*, Technical report 242816-03. Intel Corporation, Mt. Prospect, Illinois. Available from `http://-developer.intel.com/design/mmx/manuals/242816.htm`.

Jokinen, M. O. 1989. A language-independent prettyprinter. *Software—Practice and Experience 19*, 9 (September), 839–56.

Knuth, D. E. 1973. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. 2nd edition. Addison-Wesley, Reading, Massachusetts.

Kuck, D. J., Kuhn, R. H., Padua, D. A., Leasure, B., and Wolfe, M. 1981. Dependence graphs and compiler optimizations. *8th Annual Symposium on Principles of Programming Languages (POPL)* (Williamsburg, Virginia, 26–28 January 1981), pages 207–18.

LeBlanc, T. J. and Mellor-Crummey, J. M. 1987. Debugging parallel programs with Instant Replay. *IEEE Transactions on Computers C-36*, 4 (April), 471–82.

Levine, J. R., Mason, T., and Brown, D. 1992. *lex & yacc*. O'Reilly & Associates, Sebastopol, California.

Loveman, D. B. 1977. Program improvement by source-to-source transformation. *Journal of the ACM 24*, 1 (January), 121–45.

McCalpin, J. D. 1995. Sustainable memory bandwidth in current high performance computers (October). Advanced Systems Division, Silicon Graphics. This paper is available from `http://reality.sgi.com/mccalpin/papers/-bandwidth/`. Information about the STREAM benchmark is available from `http://www.cs.virginia.edu/stream/`.

McCarthy, J. 1960. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM 3*, 4 (April), 184–95.

Meyer, B. 1988. *Object-oriented software construction*. Prentice Hall, Englewood Cliffs, New Jersey.

Milner, R. 1978. A theory of type polymorphism in programming. *Journal of Computer and Systems Sciences 17*, 3 (December), 348–75.

Milner, R., Tofte, M., Harper, R., and MacQueen, D. 1997. *The definition of Standard ML (revised)*. MIT Press, Cambridge, Massachusetts.

Muchnick, S. S. 1997. *Advanced compiler design and implementation.* Morgan Kaufmann, San Francisco, California.

Nyland, L. 1994. Personal communication.

Oppen, D. C. 1980. Prettyprinting. *ACM Transactions on Programming Languages and Systems 2*, 4 (October), 465–83.

Palmer, D. W. 1996. *Efficient execution of nested data-parallel programs.* Ph.D. dissertation, published as Technical report TR97-015. University of North Carolina at Chapel Hill.

Palmer, D. W., Prins, J. F., Chatterjee, S., and Faith, R. E. 1995a. Piecewise execution of nested data-parallel programs. *8th International Workshop on Languages and Compilers for Parallel Computing* (Columbus, OH, 10–12 August 1995). Published as Huang, C.-H., Sadayappan, P., Banerjee, U., Gelernter, D., Nicolau, A., and Padua, D., editors, *Lecture Notes in Computer Science 1033: Languages and Compilers for Parallel Computing*, 346–61. Springer-Verlag, Heidelberg.

Palmer, D. W., Prins, J. F., and Westfold, S. 1995b. Work-efficient nested data-parallelism. *Proceedings of the Fifth Symposium on the Frontiers of Massively Parallel Processing (Frontiers '95)* (McLean, Virginia, 6–9 February 1995), pages 186–93, (February). IEEE Computer Society Press, Los Alamitos, California.

Parr, T. J. 1997. *Language translation using PCCTS and C++: a reference guide.* Automata Publishing Company, San Jose, California.

Partsch, H. and Steinbrüggen, R. 1983. Program transformation systems. *ACM Computing Surveys 15*, 3 (September), 199–236.

Partsch, H. A. 1990. *Specification and transformation of programs: a formal approach to software development.* Springer-Verlag, New York.

Pineo, P. P. and Soffa, M. L. 1991. Debugging parallelized code using code liberation techniques. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging* (Santa Cruz, California, 20–21 May 1991). Published as *SIGPLAN Notices 26*, 12 (December), 108–19.

Pittman, T. and Peters, J. 1992. *The art of compiler design: theory and practice.* Prentice-Hall, Englewood Cliffs, New Jersey.

Pollock, L. and Soffa, M. L. 1988. High-level debugging with the aid of an incremental optimizer. *Proceedings of the Twenty-First Annual Hawaii International Conference on System Sciences* (January 1988), volume II, pages 524–32, (January). IEEE Computer Society Press, Los Alamitos, California.

Polychronopoulos, C. D., Girkar, M. B., Haghighat, M. R., Lee, C. L., Leung, B. P., and Schouten, D. A. 1990. The structure of Parafrase-2: an advanced parallelizing compiler for C and Fortran. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, Research monographs in parallel and distributed computing, pages 423–53. MIT Press, Cambridge, Massachusetts.

Prins, J. F. and Palmer, D. W. 1993. Transforming high-level data-parallel programs into vector operations. *Proceedings of the 4th Annual Symposium on Principles and Practice of Parallel Programming (PPOP)* (San Diego, CA, 19–22 May 1993). Published as *SIGPLAN Notices 28*, 7 (July), 119–28.

Pugh, W. 1990a. A skip list cookbook. Technical report UMIACS-TR-89-72.1, CS-TR-2286.1 (June). Institute for Advanced Computer Studies, Department of Computer Science, University of Maryland, College Park, Maryland. This paper is available from `ftp://ftp.cs.umd.edu/pub/papers/papers/2286.1/`.

Pugh, W. 1990b. Skip lists: a probabilistic alternative to balanced trees. *Communications of the ACM 33*, 6 (June), 668–76. This paper (and more information on skip lists) is available from `ftp://ftp.cs.umd.edu/pub/skipLists/`.

Pugh, W. W. and Sinofsky, S. J. 1987. A new language-independent prettyprinting algorithm. Technical report TR 87-808 (January). Department of Computer Science, Cornell University, Ithaca, New York. This paper is available from `http://cs-tr.cs.cornell.edu/Dienst/UI/2.0/Describe/ncstrl.cornell/TR87-808/`.

Reasoning Systems. 1990. Refine *user's guide* (25 May). Reasoning Systems, Palo Also, California.

Riely, J. W., Prins, J., and Iyer, S. P. 1995. Provably correct vectorization of nested-parallel programs. *Proceedings, 1995 Programming Models for Massively Parallel Computers* (Berlin, Germany, 9–12 October 1995), pages 213–22, Giloi, W. K., Jähnichen, S., and Shriver, B. D., editors. IEEE Computer Society Press, Los Alamitos, California.

Rubin, L. F. 1983. Syntax-directed pretty printing—a first step towards a syntax-directed editor. *IEEE Transactions on Software Engineering SE-9*, 2 (March), 119–27.

Ruckert, M. 1997. Conservative pretty printing. *SIGPLAN Notices 32*, 2 (February), 39–44.

Smith, D. R. 1990. KIDS: a semi-automatic program development system. *IEEE Transactions on Software Engineering (Special Issue on Formal Methods) 16*, 9 (September).

Stallman, R. M. June 1993. *Using and porting GNU CC.* Free Software Foundation, Cambridge, Massachusetts.

Streepy, Jr., L. V. 1994. CXdb: a new view on optimization (8 April). Convex Computer Corporation. Unpublished manuscript. A version of this paper appeared in *Proceedings of the Supercomputer Debugging Workshop* (Albuquerque, New Mexico, November 1991).

Stroustrup, B. 1994. *The design and evolution of C++.* Addison-Wesley Publishing Company, Reading, Massachusetts.

Tip, F. 1995. *Generation of program analysis tools.* Ph.D. dissertation, published as ILLC Dissertation Series 1995-5 (17 March). Institute for Logic, Language and Computation, Universiteit van Amsterdam.

Tjiang, S., Wolf, M., Lam, M., Pieper, K., and Hennessy, J. 1992. Integrating Scalar Optimization and Parallelization. *Languages and Compilers for Parallel Computing (Fourth International Workshop)* (Santa Clara, California, 7–9 August 1991). Published as Banerjee, U., Gelernter, D., Nicolau, A., and Padua, D., editors, *Lecture Notes in Computer Science 589*, 137–51. Springer-Verlag. An overview of a more recent version of SUIF is available as R. P. Wilson, R. S. French, C. S. Wilson, S. P. Amarasinghe, J. M. Anderson, S. W. K. Tjiang, S.-W. Liao, C.-W. Tseng, M. W. Hall, M. S. Lam, and J. L. Hennessy, *SUIF: An Infrastructure for Research on Parallelizing and Optimizing Compilers*, available from `http://-suif.stanford.edu/suif/suif-overview/suif.html`.

Tolmach, A. P. and Appel, A. W. 1991. Debuggable concurrency extensions for Standard ML. *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging* (Santa Cruz, California, 20–21 May 1991). Published as *SIGPLAN Notices 26*, 12 (December), 120–31.

Wall, D., Srivastava, A., and Templin, F. 1985. A note on Hennessy's "Symbolic debugging of optimized code". *ACM Transactions on Programming Languages and Systems 7*, 1 (January), 176–81.

Warren, Jr., H. S. and Schlaeppi, H. P. 1978. Design of the FDS interactive debugging system. IBM Research Report RC-7214 (12 July). IBM Thomas Journal Watson Research Center, Yorktown Heights, New York.

Wolfe, M. 1989. *Optimizing supercompilers for supercomputers.* Research Monographs in Parallel and Distributed Computing. MIT Press, Cambridge, Massachusetts.

Wolfe, M. 1996. *High performance compilers for parallel computing.* Addison-Wesley, Redwood City, California.

Wulf, W. and McKee, S. A. 1994. Hitting the memory wall: implications of the obvious. Technical report CS-94-48 (December). Department of Computer Science, University of Virginia, Charlottesville. Appeared in *Computer Architecture News*, **23**(1): 20-24, March 1995.

Zellweger, P. T. 1984. *Interactive source-level debugging of optimized programs.* Ph.D. dissertation, published as Technical report CSL-84-5 (Xerox Palo Alto Research Center, Palo Alto, California) (May). University of California, Berkeley, California.

Colophon

This dissertation was typeset using the $\LaTeX\,2_\varepsilon$ typesetting system and the `xfig` facility for interactive generation of figures under X11. Most of the writing and program development were done under Linux and SunOS.