Pattern-Based Languages for Prototyping of Compiler Optimizers

by

Charles Donald Farnum

Abstract

Choosing an appropriate set of optimizations for a particular compiler is a black art; the literature abounds with unimplemented optimization algorithms. This dissertation describes the core tools used in Dora, an environment for exploring the design space of optimizing compilers.

Dora uses a lambda-calculus based intermediate language schema to represent programs at both high and low levels. Operators that are appropriate to a particular language, machine, and/or code-level are defined as necessary for a particular compiler, using a description language that allows the implementor to state the important properties of the operator with regard to optimization. "Machine independent" optimizations, such as moving code out of loops, are written to interrogate this information, enabling a single specification to be applicable to code at many levels for many source languages and target machines.

The language schema and description languages provide the possibility of writing optimizations in a context-independent manner, but the environment must also give special support to ease this writing. Dora includes attribution and transformation languages based on pattern matching. The pattern language is grounded in the efficient automata-driven tradition, but with important extensions for natural handling of variable-arity operators and repetitive vertical constructs such as left-associated addition trees. The attribute system uses the pattern matching system to gain the descriptive advantages of an attribute grammar system (easy access to local context and a declarative functional specification) without inheriting the difficulties of a monolithic specification factored by an often irrelevant abstract syntax. The transformation language uses the pattern matching system for local context, while relying on the attribute system for global analysis.

Dora has been used to implement a functional prototype of Frederick Chow's optimizer UOPT. The example prototype demonstrates the support Dora provides for building actual optimizers. It also makes possible previously infeasible experiments, such as reordering the optimizations, adding non bitvector-based optimizations to the suite, and applying UOPT to languages in the LISP tradition.

Contents

Acknowledgements

Fellow members of the Piper research group at U. C. Berkeley have had powerful influences on this dissertation and my growth as a computer scientist. The original conviction that the world needed some kind of compiler optimization sandbox grew out of a class project with Dain Samples, and he has continued to heavily influence the direction of this work. Eduardo Pelegrì-Llopart taught me that tree-automata are wonderful things. John Boyland, as the primary user of Dora to date, has performed a great service by asking for important new features, complaining about unusable old ones, and generally letting me know when I was being incomprehensible. He also pointed out that the DECstation compiler can produce UCODE, and thus wins the suggestion-of-the-dissertation award for making Section 6.3 possible. Chris Black suggested the name Dora; discovering the reason for this name is left as an exercise for the reader.

Sue Graham, my advisor, provided me with high goals and waited patiently for me to approach them. John Neu played the necessary and not very pleasant role of the outside-the-department member on my qualifying exam committee. Andy diSessa was kind enough to read a dissertation in a field other than his own, and dedicated enough to make serious comments after doing so. Richard Fateman and Paul Hilfinger sat as in-house members on the qualifying committee; Paul also served on the dissertation committee and suggested several improvements to the dissertation.

No dissertation was ever completed without extra-curricular assistance, and I would like to thank the members of Storyreading, the Piper group, our Monday-night Bible study, and especially my parents, wife, and children for their constant reminders that it is possible to enjoy life during graduate school.

The research described in this dissertation was funded by the State of California MICRO program, the NSF Graduate Fellowship program, and by the Defense Advanced Research Projects Agency (DoD), monitored by Space and Naval Warfare Systems Command under Contracts No. N00039-84-C-0089 and No. N00039-88-C-0292.

Chapter 1

Introduction

One of the first lessons taught in a compiler optimization course is that optimizing compilers don't optimize code. This situation will never change: producing globally optimal target code is an undecidable problem, and many sub-problems dealing with some definition of "optimal" at a local scale are NP-complete. An "optimizing" compiler is simply a compiler extended with a collection of heuristic program transformations that, at best, are proven to maintain the semantics of the program and that produce code at least as efficient as the non-optimizing compiler.

Optimizer design — choosing a good set of transformations to include in a compiler — is a difficult task. Applying transformations takes time, and so the compiler designer must choose a collection of transformations that produce good code without taking too long to do so. These transformations might be applied to the source program, the target code, or at different intermediate stages along the way; they might be applied in differing orders, or more than once. The compiler designer must take these additional factors into account, since they strongly influence the effectiveness of the transformations.

Optimizer design is not a problem that can be solved once and for all. There are many dependencies on the source language and target machines. As a trivial example, type inference combined with expansion of generic arithmetic functions can have a great influence on Lisp programs, but has little application to Pascal programs. Register coloring, a technique assuming a large number of general-purpose registers, is wasted on an IBM PC. In addition to the source language and target machine, the programming style presented by the programs to be compiled can produce more subtle effects. The influence of the language is not necessarily supreme: there are many large Lisp programs that make little use of generic arithmetic, and one can envision a substantial program in Pascal that makes heavy use of a generic arithmetic package coded with variant records.

Can the optimizer design problem be solved analytically? I claim not. The interactions arising from the ordering of the individual transformations, the levels at which they are applied, and the languages and target machines involved are already complicated enough to defy analysis. Analytic studies of the correct order in which to apply transformations, such as Chow's analysis of his UOPT optimization suite [Cho83] and Whitfield and Soffa's analysis of parallel optimizations [WhS90] generally find that optimization A should be applied before B and that B should be applied before A. Whitfield and Soffa suggest that these conflicts be resolved "based on the perceived importance of the two optimizations" [WhS90, p. 137]. But this "perception" must rely on actual experience, not on guesswork. As we will see in Section 1.3, different studies of the proper level to apply optimizations lead to the contradicting recommendations that they be applied to high-, intermediate-, and low-level code. And even if this analytic nightmare *were* solved, we would still have the problem of finding out how useful our appropriate-level correctly-ordered transformations were for the actual programs that programmers write.

Given this built-in limitation on our ability to produce an efficient and optimal compiler *a priori*, one might expect the main job of optimization research to be reaching into our bag of tricks and locating a good set of transformations for particular computing environments. Instead, most current research in optimizing compilers is algorithmic; a new algorithm is suggested for implementing a new, or oftentimes old, heuristic. On occasion, the algorithm is actually implemented as part of a compiler for a particular source language and machine, and run on enough toy benchmarks to determine that it gives a speed improvement of three to thirty percent.

Although adding to our bag of tricks is still a worthwhile endeavor, the bag is becoming increasingly unwieldy. Only a few experiments have been conducted to determine the interactions among different optimizations, languages, and machines. UOPT and ACK, two recent optimization systems with published interaction experiments [BaT86, Cho83], cover traditional global optimizations, Algol-like languages, and general-register machines. Neither system was designed with modification or extension in mind.

This is a deplorable state of affairs. Gains of three to thirty percent on toy programs written in the seventies do not impress programmers who get a machine an order of magnitude faster every few years. Analytically finding a good set of optimizations for a compiler is intractable, so we need to experiment. Unfortunately, building a particular optimizing compiler to try out a set of optimizations is currently a very expensive project.

1.1 Optimizers are hard to build

We need to experiment, but this experimentation is difficult. Why is it so hard to write optimizing compilers? There are several reasons:

Big projects. A full compiler for a real language, even without optimizations, is a large piece of software. When optimizations are added, the problem gets worse. Putting together a compiler is a massive effort; it is very easy to get swamped by too many simple details.

Complexity. Many individual parts of the compilation process are relatively simple, but they become complicated when they are merged into one pass. This happens frequently within an optimizer, where the implementor may do many optimizations at once in an attempt to make the most of hard-to-recompute analysis data and/or to speed up the optimization process. In addition to this merging effect, many optimization algorithms are inherently complicated, often to the point that they are discussed, analyzed, and "improved" in the literature without ever actually being implemented.

Efficiency-oriented. Optimizing compilers are often heavily optimized at the source-level, by the implementor, in an attempt to speed up the compilation process. Optimization of the compiler primarily shows its effects by making the resultant code both more complicated (via merging of passes and/or an overuse of low-level language features) and larger (by avoiding stylistically appropriate but practically inefficient abstractions).

Lack of tools. There are an abundance of tools aimed at various stages in the compilation process, including scanning, parsing, static-semantic analysis, and code generation. But there is a marked lack of useful tools for implementing optimization analyses and transformations. Several researchers, including the members of the OPTRAN and Ergo projects [LMW88, NoP88], have made attempts to use attribute grammars for analysis, but these systems are much less effective at their tasks than the corresponding front-end and code-generator tools. Without suitable global analysis tools, tree transformation systems have been used primarily in other areas of the compilation process.

Many of these problems can be solved through *prototyping* the compiler — building a compiler whose sole purpose is to provide a testbed for the optimization suite. In a prototype, the need for the efficiency-orientation is greatly diminished. The compiler still needs to run fast enough to perform experiments, and there must be some path from the final prototype to an efficient production version, but we can make great strides in reducing the other problems by relaxing the efficiency constraint. In a prototype, more use can be made of reusable parts, without the need to tailor code to run efficiently in a particular situation. Large problems can be factored into smaller ones with less concern over the phase-boundary inefficiencies introduced in the process. Complicated tasks can be made easier through the use of high-level languages, reuse of complicated sub-tasks, and elimination of the need to perform several transformations in one pass.

But a simple shift in emphasis from building production compilers to prototype compilers is not enough. Writing code that is reusable in compilers for varying source languages and machines is itself a non-trivial task, and requires some commonality among the different compilers and their internal representations. Better tools are needed to support the tasks frequently performed in optimization building. In this dissertation, I describe Dora, an environment that supports experimentation with optimizing compilers by making it easier to build, rearrange, and reuse optimizing transformations.

1.2 Desiderata

Before outlining the desirable features for an optimizer-prototyping environment, let us first delineate the scope of an optimizer and define some terms. Dora assumes the traditional division of compilers into two main units, the "front end" that deals with lexical, syntactical, and static-semantic analysis of a program, and the "back end" that uses the information gathered by the front end to generate efficient machine code. In an optimizing compiler environment intended for multiple source languages, the front end's job is to correctly translate a source program into an *intermediate representation* (IR) of the program in some appropriate *intermediate language* (IL); it is the optimizer's job to convert this intermediate representation into a suitably efficient body of assembly code. An optimizer performs this conversion by applying two different kinds of transformations to the IR: *vertical* transformations that take a program p_1 in some language L_1 and transform it into an "equivalent" program p_2 in a different language L_2, and *horizontal* transformations that take a program p_1 in language L_1 and transform it into a better program in the same language.

In order to qualify as a general-purpose environment supporting experimentation with optimizing compilers, Dora must fulfill the following criteria:

- It must be possible to create compilers for a wide range of languages. In particular, there must be strong support for both traditional Algol-like languages (which cater to stack implementations and static typing) and traditional Lisp-like languages (which require first-class closures and dynamic typing). Implementing "logic" languages (such as Prolog), functional languages, and parallel languages should also be possible. A system restricted to Algol-like languages is insufficient, given the growing theoretical and practical importance of other programming paradigms.

- It must be possible to create compilers for a wide range of machines. There must be strong support for traditional general register machines, as the still-dominant architecture, but vector machines and parallel machines should not be ruled out.

- Adding languages and machines similar to those already implemented should be a matter of days or, at most, weeks. Without this capability, Dora will become a testbed for building compilers for fixed language/machine combinations; a useful tool, but not the one we desire.

- Implementing optimizations must be made as simple as possible. In particular, there should be a powerful transformation language for easily expressing the result of a particular optimization. A library of abstract data types commonly used in optimization algorithms must be present. A data-flow analysis tool is required.

- Implementations of particular optimization techniques must be as reusable as possible. An optimization that could be used in many different contexts — for different languages and machines, applied to the source language, machine code, or levels in-between — should need to be implemented only once. Dora must support writing optimizations, to whatever extent is possible, in a source-language/target-machine/code-level independent manner.

1.3 Related systems

Given the amount of research activity involved with "programming environments" and "compiler generation", it is somewhat surprising that a system fulfilling the criteria stated for Dora does not exist. In fact, there are relatively few optimizing compilers known to the research community that support both multiple source languages and multiple machines. This section briefly describes the major efforts that have influenced the design of Dora.

1.3.1 ECS

The Experimental Compiling System (ECS) [ACF80] was an early attempt at supporting multiple languages and machines by using an intermediate language (IL) *schema*. In a traditional IL, the set of operators is fixed. This causes problems for a general system: optimizations of the IL can only be applied at one level (the level of the operator set), and the chosen operators can severely limit the source languages and/or target machines that can be handled. An IL schema provides a set of primitive operators and a definition mechanism for adding new operators to the IL, overcoming these difficulties.

In ECS, a powerful optimizer was intended to operate using only information built into the schema and attributes declared for the operators. The definitional mechanism for new operators was the procedure abstraction. The system was not a compiler-compiler; a front end and back end were assumed to exist. Rather, the system was an attempt to provide a powerful optimizer that would convert a naively generated program in a high-level IL into an efficient program in a low-level IL through procedure integration. The project created two large design papers and a few technical reports before dying a quiet death.

Members of the ECS group claim that the use of PL/I as the primary source language swamped the project with too many details [Sam87]. The procedure-integration paradigm was also believed to be too slow for other immediate concerns of their research community.

1.3.2 PL.8

The PL.8 system [AuH82] was created by a research project at IBM in the early 80's. Front-ends for Pascal, PL.8 (a systems oriented variant of PL/I), and C exist; the compiler generates code for several 32-bit byte-addressable machines. In PL.8, front ends naively generate a low-level IL similar to the instruction set of the 801 (IBM's first RISC machine). The system applies several traditional global optimizations to this low-level code, assigns registers using register coloring, and then generates machine-code using a simple instruction selector.

Although less ambitious than the ECS project, PL.8 does use one of the ideas from ECS; although basically fixed, the operator set of the IL can be extended with special instructions that will later be expanded into other more primitive instructions. The properties of these instructions can be defined for use by the optimization algorithms, thus giving the ability to optimize at two levels; once before expansion, and once afterwards. This facility is used, for example, to allow common sub-expression (CSE) removal of the MAX function; after an optimization pass has been applied, the MAX operator is then expanded to a series of tests, branches, and stores that the CSE algorithm is too weak to recognize.

1.3.3 UOPT

Frederick Chow wrote UOPT [Cho83], a global optimizer for a fixed IL called UCODE, at Stanford in 1983. UOPT is intended to be used for multiple languages and multiple targets; a language-specific front-end generates naive UCODE, UOPT optimizes the UCODE and performs register assignment, and a machine-specific back-end generates machine-code from the optimized UCODE. UOPT is parameterized by a machine description that states the relative costs of various primary functions, for example, loading a register.

Chow's dissertation includes experiments reporting the effectiveness of different combinations of optimizations for different source languages and target machines. His experiments yielded the information that different targets received differing benefits from various combinations of optimizations, supporting our claim that such experiments are necessary. Surprisingly, he did not experiment with different orderings of the optimizations, but instead argued extensively for the particular order that was implemented. UOPT is not suitable as a general tool; it is severely restricted by its limited intermediate language, which was originally designed as an abstract-machine style target code for Pascal. In addition, it provides no support for the addition of new optimization techniques; it was built for efficiency, not mutability.

1.3.4 ACK

ACK, the Amsterdam Compiler Kit [BaT86], is a set of tools used to build optimizing compilers for block structured languages that can be implemented with a stack. The optimization portion of the compiler is a set piece of code. The main interest of the creators was to determine machine parameters that would allow easy retargeting of traditional optimizations. ACK is primarily interesting (from our point of view) in that the wide range of optimizations provided are driven by a small machine description. Its IL is slightly more general than UCODE, but it is still too weak to handle languages with first class functions, and it is not clear that adding new optimizations would be aided by the fact that some are already implemented.

1.3.5 OPTRAN

The OPTRAN system [LMW88] performs transformations on attributed trees. It is a descendent of the MUG2 attribute grammar system [GGM82]. In OPTRAN, attributed syntax trees are used as the intermediate representation. Optimizations are written as tree transformations; the writer of a transformation can state that it leaves certain attributes unmodified, can provide update routines for other attributes, and state that others must be recomputed.

The transformation language in OPTRAN is quite similar to that used in Dora. The major differences between OPTRAN and Dora lie in OPTRAN's emphasis on producing efficient compilers and Dora's emphasis on prototyping. Since OPTRAN uses syntax trees for its intermediate representation, transformations cannot be reused from one compiler to the next. OPTRAN's reliance on efficient attribute grammar technology hampers its ability to prototype compilers quickly, as I will argue in Section 4.1; for example, it is impossible to define attributes with circular dependencies, which arise in data-flow-problems for languages with GOTO.

1.4 The Design of Dora

Dora has been designed to satisfy the desiderata, taking into account lessons learned from the review of multi-language, multi-target optimizing compilers.

1.4.1 The Intermediate Language Schema

We require that optimizations that have the potential to be applied at many different levels (for example, CSE elimination) need be implemented only once, yet apply to all levels. This is a problem in UOPT, ACK, and the PL.8 compilers; all three do most of their optimization at one level, and each argues that their particular level is the right one (ACK uses a relatively high-level stack oriented IL, UOPT uses an intermediate level with both a stack and registers, and PL.8 uses a very low level IL similar to a RISC machine's instruction set).

The ECS system reveals part of the solution: by using an IL schema, different levels of code can be represented in a uniform way. Optimizations that depend only on the information that can be described with the schema can be implemented in a level-independent way. For example, performing the CSE optimization requires examining the control flow, side effects, and uses of the store in a program, but it does not require knowledge of whether a given operator is performing a set intersection in Pascal or a longword addition in assembly language. With an appropriate schema, a CSE transformation can be written to be equally applicable to an assembly language program and a program in Pascal.

In order to make possible the implementation of widely-applicable optimizations, Dora uses a single IL schema to represent programs at all levels. The IL representation of a program is the only carrier of semantic information; an optimization must be able to function correctly given the intermediate representation output by any other phase. This places a strong burden on the intermediate language. It must be general enough to represent many different source languages and machines, yet have enough structure that optimizations can be written without knowledge of the particular language involved.

DILS, Dora's intermediate language schema, is based on the lambda-calculus extended with a store. DILS is discussed at length in Chapter 2.

1.4.2 Describing the source language

We require that it be easy to add new source languages, and that extant optimizations can be applied to the new languages. Dora takes the existence of a semantically-checked abstract syntax tree for granted; a minimal description of a source language then consists of a transformation from semantically checked abstract syntax trees to a high-level DILS IL. Implementing these transformations is made simple through two features of Dora: the ability to define operators in DILS that are equivalent to the operators and control structures of the source language, and a pattern-based tree-transformation language. By defining appropriate DILS operators, the transformation can be specified by a simple listing of the source-language constructs with their equivalent DILS implementations.

It is the responsibility of the front-end implementor to define operators in DILS corresponding to operators in the source language. Initial languages in a new application area may require large amounts of work as a suitable set of DILS operators is defined, but future languages needing similar operators can then make use of the existing definitions with only minor changes.

This "description" of the source language through the source-to-DILS transformation might be seen as a purely operational one, as it consists of an implementation of the language in terms of DILS. But a simple implementation of the source language is not sufficient for an effective optimizing compiler; some additional information about the source language must be provided. This information falls into two general classes:

- Assumptions made by the front-end about the context in which particular functions will be executed. For example, a Pascal implementor might state that functions are never stored in variables nor returned as results.
- Facts about particular bodies of code needed for certain optimizations. The facts may be either difficult or impossible for Dora to determine from analysis of the code. For example, the HP Spectrum compilers make use of source-language specific type information to help obtain accurate aliasing information [CHK86]. Rather than requiring the optimizer to understand the various type systems, the Spectrum front ends are required to encode basic aliasing patterns in the IL.

Information of this nature can be provided in the source-language to DILS transformation in two ways. Global information about procedures is encoded in the *calling convention*, a symbol associated with each procedure that is used by further transformations in Dora. Information that might be encoded in the calling convention includes, for example, the fact that arguments for a Pascal procedure can be allocated on the control-flow stack, rather than on a more general heap. The set of calling conventions is extensible at compiler-building time, but not at compile time, so a different mechanism is needed to provide fine-grain information such as the aliasing information in the Spectrum compilers. Fine grain information can be directly included in the IL code by providing suitable pseudo-operations and operands.

1.4.3 Machine descriptions

We require that it be easy to add new target machines, and that extant optimizations apply to new machines. As with source languages, this implies the existence of a machine description. Machine descriptions are more important to Dora than source language descriptions; much of the information about a source language is captured in the process of translating the source language into the procedural part of the IL, which is outside the scope of Dora's work, whereas translating the IL into the machine's assembly language is part of Dora's job.

No optimization is completely machine-independent, and it is typical for optimizers to make use of a machine description that states the relative cost of certain common operations. Dora has a simple "set of parameters" description of the machine for this purpose. In addition to this information, Dora also needs a more complete description of the machine's instruction set in order to generate assembler code. Dora uses the tree transformation techniques developed by Pelegrí and Farnum [Pel88] [Far88], which provide an excellent instruction-selection facility. The machine description consists of a tree transformation from a low-level DILS IL to a DILS IL with the machine instructions as an operator set.

1.4.4 Transformation languages

Transformations play a major role in Dora; they assist in describing both the source language and the target machine, in addition to their use in optimizing the code. Special language support is needed to ease the writing of both horizontal and vertical transformations on DILS. Dora provides Tess, a transformation system based on rewrite rules. A rewrite rule consists of an input pattern and an output pattern. The input pattern restricts the DILS structures to which the rule is applicable; the output pattern shows how to construct the transformed structure out of the pieces of the input.

Rewrite-based systems are very popular in compiler systems with tree-structured intermediate languages. There are two main styles currently in use, automata-based tree transformation systems and general pattern-matching languages. In an automaton-based system, the set of rewrites is analyzed to produce a table that enables finding all matches of the input patterns in a single pass over the tree; depending on the complexity of the rewrite system, it is sometimes possible also to precompute the effects of multiple rewrites at a single node, or to apply simultaneously two conceptually separate rewrite systems. These systems are fast, but have a limited pattern-matching language. General pattern-matching languages, on the other hand, provide powerful patterns (often with the capability to escape into some base language such as Lisp) but give slow pattern matchers and usually have no special control structures to apply a set of rewrites efficiently throughout a tree.

Tess is based in the automaton camp. General pattern-matching systems are powerful, but they tend to be *ad-hoc*, with unpredictable execution times for similar patterns. Although Dora is intended for prototyping, and fast compilation is not a high priority, linear run-time guarantees of the transformation primitives are vital to keep experimentation possible. In order to provide the power needed for optimization transformations, Tess both extends traditional automata-based patterns and allows the use of semantic preconditions to limit the applicability of individual rewrites. Chapter 5 describes Tess and its implementation.

1.4.5 Semantic attributes

The input pattern of a rewrite rule restricts the application of a transformation based on local structure. In optimization algorithms, it is frequently necessary to take global information into account as well.

In compiler research, attribute grammars are a popular method of propagating global information throughout trees. Dora's semantic-attribute language Sal provides many of the advantages of attribute grammars while avoiding one of their main drawbacks, the reliance on a monolithic grammar to provide the case analysis for all attributes. Instead of a single abstract grammar, Sal uses patterns tailored to individual analysis problems. Abandoning the grammar also entails abandoning the body of attribute grammar evaluation techniques; Sal uses algorithms based on dynamically-discovered dependencies and worklist construction to allow the solution of recursive attributes. Chapter 4 describes Sal and its implementation.

One particular kind of global information, data-flow analysis, has received particular attention in the past due to its usefulness in optimization. There is a vast literature on data-flow analysis of traditional intermediate representations using the control-flow graph (CFG). Lambda-calculus based ILs do not have an explicit CFG, but the CFG based algorithms can be used by uncovering the implicit control flow in the code. Olin Shivers discusses how to perform the necessary analysis to discover the underlying control flow [Shi88]. Data-flow analysis in Dora is currently performed by using Sal attributes to produce a control-flow graph, as described in Chapter 6. Douglas Grundman is investigating an alternate analysis language designed specifically for solving data-flow problems [Gru90].

1.5 Currently implemented systems in Dora

Experimenting with optimizing compilers is a largely unexplored research area, and the design of Dora encompasses many separate research topics. My dissertation research has concentrated on the design of a suitable intermediate language schema and on attribute and transformation languages based on pattern matching, and on the implementation of these languages in Common Lisp. The languages of Dora are *embedded* in Common Lisp; they are intended to be used as extensions of the Common Lisp language, rather than introducing a completely new syntax and duplicating facilities in Common Lisp useful in prototyping. The user of Dora writes an optimizer in Common Lisp, using the Dora extensions when appropriate. The extensions can be categorized into four main components: support for the intermediate language DILS, a tree-pattern-matching language, the analysis language Sal, and the transformation language Tess. These tools are described in detail in Chapters 2–5.

Since Dora is implemented in Common Lisp, many of the examples make use of Common Lisp code and concepts. Appendix B contains a brief glossary of Common Lisp terms for the reader unfamiliar with Common Lisp, along with a glossary of the special forms defined by Sal and Tess.

1.5.1 DILS

Chapter 2 defines DILS, Dora's intermediate language schema, which unifies the different intermediate languages used within Dora and makes possible the implementation of language-independent optimizations. A simple internal representation of the intermediate language schema constructs is implemented. In addition to the typical construction and access functions, a macro `build-dils` is provided that constructs the internal representation of a DILS expression and performs type-inference, given the expression's printed representation. `build-dils` can be used by the prototyper to construct test portions of code, to build front ends, and to build new code pieces during the transformation process.

1.5.2 Support for pattern matching

Chapter 3 describes the pattern-matching language used in Dora. It provides background on tree-pattern-matching systems, describes two new kinds of patterns — horizontal and vertical iterators — and describes an efficient matcher for the new patterns. A macro `tree-match-case` has been implemented that makes it easy to use the pattern-matcher outside of the special forms of Sal and Tess. `tree-match-case` takes a tree and a list of pattern/action pairs, and performs the given actions at nodes in the tree matching the corresponding patterns. It is useful for debugging, or for occasional analysis problems or transformations that do not map well onto the Sal/Tess models.

1.5.3 Sal

Chapter 4 describes Sal, a new attribute language based on the pattern matching work. *Attributes* are properties of tree nodes that are functions of the current tree. They are specified by describing a set of *attribute equations*, which define an attribute value at a given node in terms of attributes of other nodes. Node variables in the attribute equations can be restricted by pattern matching or by semantic checks, that is, by tests on other nodes and their attributes. Attributes are defined with the `def-attribute` form, which creates a Lisp-callable function to access the value of the attribute given a node. This function recomputes the value of the attribute when necessary.

1.5.4 Tess

Chapter 5 describes Tess, a new transformation language based on the pattern matching work. In Tess, transformations are specified by describing a set of possible node replacements, which pair a node with an *instantiation pattern* describing the construction of a new node. The node to be replaced, and the variables in the instantiation pattern, are restricted by pattern matching and semantic checks, just as in Sal. Tree transformations are defined with the `def-transformation` form, which creates a Lisp-callable function that takes an input tree, applies the node replacements in an order that can be controlled by the transformation definition, and returns the transformed tree.

1.5.5 The UOPT prototype

Chapter 6 describes my experience constructing a prototype of Chow's UOPT optimizer. The prototype defines DILS operators similar to the UCODE instructions used in UOPT, defines Sal attributes that perform the local and data-flow analyses required by UOPT, and defines Tess transformations that make use of the Sal attributes to determine how to transform the DILS trees. A simple Lisp function serially applies the Tess transformations to the DILS internal representation to produce the optimized output.

1.5.6 Conclusion

The ease of prototyping the UOPT system demonstrates that Dora provides assistance in experimenting with compiler optimizers. Chapter 7 discusses how well DILS, Sal, and Tess meet the desiderata of Dora, and describes where research with Dora is headed.

Chapter 2

DILS

There are two primary design considerations pulling in opposite directions on the choice of an intermediate language for Dora. As a system for prototyping any of a number of different final compilers, the representation should be very general, allowing the prototyper to implement compilers for widely different languages and machines, and modeling different styles of IR use. As a system for encouraging the reuse of existing components, the representation should be semantically rich and detailed; if the representation is too general, individual prototypers will add the detail necessary for their particular compiler, and any optimizations they write will either be over-parameterized or dependent on the additional detail.

DILS, the intermediate language schema used in Dora, provides a very general system for modeling particular representations while simultaneously giving a rich set of semantic primitives. DILS provides the power of the lambda calculus for describing control flow and a basic store for handling side effects. In this chapter, we examine the syntax and semantics of DILS through the use of examples, seeing how DILS can be used to model different popular styles of IR and how it can represent programs at many different levels, all the way from the source language down to the machine. A concise definition of DILS is given in Appendix A.

2.1 Primitive operator application

Operator application is at the heart of most intermediate representations. An *operator* is a primitive function taking a fixed number of arguments and yielding a result, possibly modifying a store as a side-effect. In a primitive application, the arguments are all constants of some kind, either constant values to be manipulated or constant store locations to be used or modified.

Most of the "statements" in a typical intermediate language are primitive operator applications. For example, in the assembly language statement `mov r1,17`, the `mov` operator is applied to the constant arguments `r1` and `17`. The *quad*, a common intermediate form, consists of an operator and three arguments; the operator computes some value from the first two arguments and stores the value in the third argument.

Primitive operator applications are represented in DILS by enclosing the operator and its arguments within parentheses. The assembly language statement could be written in DILS as `(mov r1 17)`, and a typical quad as `(plus a 3 b)`.

The choice of an appropriate set of operators and constants is often seen as the most important job of the intermediate language designer. In emphasizing the operators of the language, designers choose a particular level for the IL; operators may be defined similar to the operators of some source language, to the operators of a particular machine, or to some set of idealized operators in the middle. As noted earlier, the designers of ACK, UOPT, and PL.8 all argue for their different choices of the "right" level to define their operators and perform most of their optimization. The members of the ECS project, on the other hand, wanted to perform optimization at all levels; they proposed fixing a structure for the IL (quads, for example), but leaving the definition of operators to some specification language. Optimizations could manipulate the IL through knowledge of its structure and by querying properties of the operators that were defined in the specification language. Given such an IL *schema*, the individual operators become less important; new operators can be defined at will, so long as their important properties (from an optimization and implementation viewpoint) are specified.

An extensible operator set is clearly a basic requirement for Dora, given our goals for handling widely differing source languages and applying optimizations at many different levels. Perhaps less obvious is the need for an extensible set of constants, but this becomes clear when one considers the need to handle different floating-point formats, describing machines such as Lisp machines with language support for more esoteric data types, and the desirability of describing high-level constants such as Pascal's `Nil` in a machine-independent fashion. Dora provides three different kinds of basic operators and constants: integers, character strings, and *atoms*. Integers are represented by digit strings (`42`), character strings by strings bracketed with double quotes (`"This is a character string"`), and atoms by non-integer alphanumeric strings (`This-is-an-atom`). Atoms are defined via a specification language, and usually represent operators although they can also represent memory locations (like `r17`) or other types of constants (like `Nil`). The specification language is described in Appendix A.

2.2 Basic blocks and simple control-flow

In most common ILs, including the various kinds of tuples, register transfer languages, and many tree-expression ILs, individual statements are grouped into straight-line sequences of statements. These sequences are called *basic blocks*, and form the primitive unit for many optimization algorithms. The last statement of each basic block is some form of transfer statement, describing the possible blocks to be executed next; thus, the basic blocks form the nodes of a control-flow graph. A typical basic-block representation of a while loop is illustrated in Figure 2.1.

In order to represent basic-blocks, we need to be able to handle sequences of primitive operator applications. This can be done in DILS by using nested applications. The arguments of a DILS application are not restricted to be constants, but can be any expression, including another application. Once the children of an application can be non-constant expressions, order of evaluation becomes an important issue; in DILS, the order of

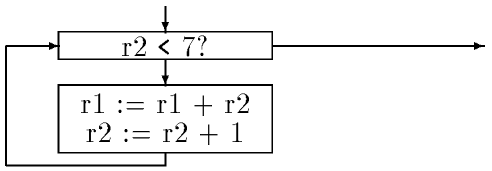


Figure 2.1: A simple while loop in basic-blocks

evaluation is specified as left-to-right. Thus, the body of the while loop in Figure 2.1 can be represented by the DILS application

```
(e2 (add-to r1 r2)
    (add-to r2 1))
```

with the `add-to` operator defined to add the second argument to the first, and the `e2` operator defined to simply return the value of the second argument. By building up several `e2` applications in a tree, an arbitrary sequence of sub-expressions can be executed in order, much as the `CONS` operator is used to build up lists in LISP.

By using sub-expressions, we can construct a sequence of statements, but we do not yet have the power to build conditional execution. If we tried to model an if statement, for example, with the application `(if` *condition stmt1 stmt2*`)`, we would be doomed to failure regardless of the definition of the `if` operator; the statements, as subexpressions of an application, would both be evaluated before the `if` operator gained control. What is needed is some way to prevent the evaluation of an expression, and treat it as a code object to be manipulated (and possibly evaluated in the future).

Evaluation of expressions can be delayed in DILS with the *thunk*[1] form. The thunk form is written as `(th` *expression*`)`, and returns a function that, when applied, evaluates the *expression*. The function might be applied either by some operator, or explicitly by appearing as the first element in an application list. For a trivial example, `((th` *expr*`))` is equivalent to *expr*. There is an appropriate definition of the `if` operator when we use applications of the form `(if` *condition* `(th` *stmt1*`) (th` *stmt2*`))`; the `if` operator needs to apply its second or third argument depending on the value of *condition*. A basic-block, then, can be represented easily in DILS by a thunk form with the statements of the block connected via applications of the `e2` operator. We could represent the basic blocks of Figure 2.1 with the thunks

```
test-thunk =  (th (if (< r2 7)
                      body-thunk
                      end-thunk))
body-thunk =  (th (e2 (add-to r1 r2)
                      (e2 (add-to r2 1)
                          (test-thunk))))
```

We could now construct the basic block representation by creating a circular structure. But there are many reasons to argue against an implicitly circular structure in an intermediate

[1]The term *thunk* was coined by Ingerman [Ing61]. It is used here to denote any parameterless procedure.

representation; they are difficult to represent textually, graph manipulation languages are much more difficult to write and read than tree manipulation languages, and finally locations where circularity might arise are important in and of themselves in optimization, and should be explicitly noted. For all these reasons, DILS is a tree-structured representation. We therefore need to have some way to refer indirectly to thunks in order to handle circularities; this is done with the *labels* form. Our while loop example is written with the labels form as follows:

```
(labels
  ((test (th (if (< r2 7)
                 body
                 end)))
   (body (th (e2 (add-to r1 r2)
                 (e2 (add-to r2 1)
                     (test)))))
   (end  (th *void*)))
 (test))
```

The labels form takes a list of label/thunk pairs, and a sub-expression; within the thunks and the sub-expression, the labels are bound to the corresponding thunks. The labels form is evaluated by evaluating the sub-expression. In our example, evaluation begins by evaluating `(test)`; this evaluates the `if` expression, which evaluates either the `body` thunk or the `end` thunk, and so on. The `end` thunk simply returns the constant `*void*`, used as the canonical meaningless value to be returned when using an expression as a statement.

2.3 Functions

Typical basic-block oriented ILs become *ad hoc* at the procedure level; a procedure is represented by a set of basic blocks and various tables describing the parameters, local variables, and other properties of the procedure. In many languages, especially languages encouraging the creation of many small functions, this abrupt break in the IL is inappropriate. DILS represents functions with the *lambda* expression, based on the lambda abstraction of the lambda calculus.

A lambda expression, when evaluated, yields a function taking a fixed number of arguments. Its syntax is `(lm (`*parms*`)` *body*`)` where *parms* is a list of symbols used as the formal parameters of the function, and *body* is an expression that is evaluated when the function is applied. For a simple example, `(lm () (if (< r2 7) body end))` evaluates to a function that, when applied to zero arguments, will apply either `body` or `end` depending on the value of `r2`. A thunk is equivalent to a lambda expression with no parameters; `(th` *body*`)` is simply shorthand for `(lm ()` *body*`)`. The `labels` form, in addition to accepting thunks, also allows arbitrary recursive functions to be defined.

In the lambda calculus, applying a function binds the formal parameters to the corresponding (evaluated) arguments in the application. As part of DILS support for imperative programming languages, a DILS formal parameter is instead bound to a *cell*, a modifiable information holder, which is initialized to hold the corresponding evaluated argument. The formal parameter may then be referenced within the body of the function in two different ways: `^`*var* is an expression that evaluates to the current contents of the cell bound to *var*, and `&`*var* is an expression that evaluates to the cell itself. This additional notation (an alternative would have been to have *var* refer to the cell, and use operators such as `fetch` to obtain the value) sometimes proves convenient when analyzing the code; for example, variables that are only referenced with the `^`*var* form can be treated as regular lambda-calculus variables.

The contents of a cell may be modified by operators such as `assign`, which takes a cell and a value and stores the value in the cell.

```
(lm (x)
  (e2 (assign &x (plus ^x 1))
      ^x))
```

is a simple function that takes a single value, adds one to it, and returns the incremented value. The parameter passing is by value, since all arguments to an application are always evaluated before the function is applied;

```
((lm (y)
     (e2 ((lm (x) (assign &x (plus ^x 1)))
          ^y)
         ^y))
  7)
```

returns 7, not 8. Call-by-reference may be achieved by passing the cell bound to a variable, rather than its value; call-by-name is achieved through passing thunks, as we have already seen with the `if` operator.

Parameters may be referenced anywhere within the body of their corresponding lambda expression; in particular, they may appear within nested lambda expressions. These references are lexically scoped, as in Scheme; when a lambda expression is evaluated to yield a function, a closure over the free variables in the expression is created.

2.4 Continuations

The addition of the lambda expression enables us to represent functions cleanly in DILS, rather than relying on some *ad hoc* construction. But even with lambda expressions, many functional ILs resort to trickery when it comes to representing non-local transfers of control, such as the `catch/throw` construct in Lisp. These non-local transfers can also occur in traditional imperative programming languages, for example, using `goto` to exit a procedure in Pascal. Non-local control transfers can be modeled cleanly through the use of *continuations*.

Consider some execution of a program just after a given function application. The application has returned a value; the remainder of the computation can be viewed as a function of the current store and the value returned. This functional abstraction of the remainder of the computation is called the current continuation. Non-local control transfers are modeled by replacing the current continuation with a different continuation, one that correctly describes the desired "remainder of the computation".

Steele [StS76] shows how continuations can be used to implement every sequential control construct known in 1976, and his examples are easily extended to control constructs introduced since then. By "sequential" control constructs, I mean control constructs that can use at most one processor at a time; co-routines, for example, are a sequential control construct since only one routine is active at any give time. Steele introduces *continuation passing style*, or CPS, in which the constructs we have already seen (particularly lambda expressions) are used to model continuations explicitly. In CPS code, each procedure is passed an explicit function representing the "remainder of the computation". Instead of "returning" a value, the procedure is required to pass its return value to the explicit continuation it was passed.

CPS code has some advantages for an optimizer, particularly in a Scheme compiler. David Kranz et al. [KKR86] and Shivers [Shi88] point out the advantages of a very simple syntax and the ability to treat many control-flow problems as data-flow problems. But there are disadvantages as well. CPS code is very difficult to read, and thus makes debugging the optimizer more difficult. The "simple syntax" can actually be a disadvantage for many circumstances, since it destroys the natural tree structure of the code; producing CPS code from abstract syntax trees cannot be done with simple tree transformations, and code generation techniques based on tree structured ILs cannot be applied easily. The tree-oriented analysis and transformation languages within Dora favor maintaining natural tree structures when possible. For all of these reasons, DILS has special forms to create and apply continuation objects. CPS code can, of course, be used in a particular Dora compiler by restricting the code to a subset of DILS and enforcing the continuation passing conventions.

Continuations in DILS are first-class objects; they may be passed as parameters, returned from functions, and placed in the store. DILS provides two kinds of expressions for dealing with continuations. The *continuation capture* applies a function to the current continuation, thus making the continuation available as the value of a formal parameter. It is denoted by the form `(c-cap` *func*`)`. The *continuation application*, denoted with the form `(c-app` *cont val*`)`, replaces the current continuation with the value of the *cont* subexpression. The new continuation is passed the value of *val* as an argument. Continuation captures in DILS are quite similar to the `call-with-current-continuation` form of Scheme; the primary difference is that the continuation created by Scheme is called just like an ordinary closure, while a DILS continuation requires the `c-app` special form for application. The use of the special form makes it easier for compilers that choose to make limited use of continuations to do so, since they can safely assume that normal application forms will apply normal functions, not continuations. The separate special form is also semantically cleaner: a continuation application, which discards the current continuation, is quite different from the normal function application, which maintains the continuation.

A function that finds the product of the integers in a list, returning zero immediately if some element is zero, is defined in Figure 2.2. Lambda expressions in the figure have been subscripted to aid in the following trace of this function.

$\texttt{lm}_1$ takes one argument, `int-list`. The body of the function applies $\texttt{lm}_2$ to the argument `1`; this is a simple way of creating and initializing a local variable `prod`. The continuation capture expression binds `return-from-lm-3` to a cell holding the current continuation, which at this point simply returns from $\texttt{lm}_2$ and $\texttt{lm}_1$.

The body of $\texttt{lm}_3$ computes the proper value of `prod` with a `while` loop, and then

```
(lm1 (int-list)
  ((lm2 (prod)
     (c-cap (lm3 (return-from-lm-3)
              (e2
               (while (th4 (not (empty-int-list ^int-list)))
                 (th5
                   (e2
                     (when (zerop (int-list-car ^int-list))
                       (th6 (c-app ^return-from-lm-3 0)))
                     (e2
                       (assign &prod (times ^prod (int-list-car ^int-list)))
                       (assign &int-list (int-list-cdr ^int-list))))))
               ^prod))))
   1))
```

Figure 2.2: Finding the product of a list of integers in DILS

returns this value. The while operator takes two thunks, the first acting as the loop test and the second as the loop body. In the loop body, the `assign` applications appropriately update `prod` and traverse the int-list. The `when` operator is like an if-statement with only one branch; the branch, th$_6$, is executed when the condition is true. If the first element of the `int-list` is zero, `(c-app ^return-from-lm-3 0)` will be evaluated. This continuation application replaces the current continuation with the one stored in `return-from-lm-3`, passing it zero; control immediately is transferred to the point after the continuation capture, with the capture expression returning the value zero.

Abelson and Sussman provide an up-to-date and thorough discussion of continuations and their use in modeling different control-flow constructs [AbS85].

2.5 The type system

The main constructs of DILS are most similar in spirit to ILs in use in the Lisp community. Lisp systems that have used a lambda-calculus based IL have avoided typing the intermediate code. When the source language is dynamically typed, this causes few problems; a function call has to be prepared to accept an object of any type, and there are run-time conventions that assure a standard method of passing objects using a fixed amount of space.

With statically typed languages, an untyped IL is problematic. Suppose we wish to have separately-compiled modules with cross-module function calls. In order to have a non-generic parameter passing mechanism, we need to have information provided by the static type of the function parameters. If the IL is untyped, this information must be embedded in the IL code. This embedding is workable, and has been used, for example, by Richard Kelsey in his transformational compiler [Kel89]. Unfortunately, this embedding of type information in the code makes it more difficult to write simple transformations;

the type information can usually be ignored by optimizations, and so should be implicit, rather than explicit, in IL. Explicit type information in the IL quickly leads to bulky and hard-to-read transformations.

DILS uses a static type system to provide information necessary for parameter passing without forcing it to be explicitly mentioned when manipulating code. In addition to its uses for parameter passing, type information has also proved useful in two other areas: first of all, it provides a convenient repository for general declarative information provided by the front end (e.g., the type of a function might include the information that it can safely use stack-based allocation for its variables), and secondly, it provides a safety check for transformation writing; when a subtree is replaced, the resulting tree must still type-check.

A *type* in DILS is similar in structure to a Lisp s-expression. There are two kinds of types, *primitive* and *composite*. A primitive type is denoted by a symbol. A composite type is composed of a pair of other types, denoted with Lisp's cons-cell notation. Thus, `integer` and `string` are primitive types, while `(fn prim integer integer)` is a composite type. The meaning of a particular type is provided by the transformations used in a particular compiler to generate code, and is not currently fixed in Dora, just as the set of operators is not fixed; the user of Dora can add new primitive types (and, by construction, new composite types as well) appropriate for the operators and data types of a particular language or machine.

In a typical statically-typed imperative language, types are declared for all procedures and variables. In a functional language, the preponderance of lambda expressions makes such a type system very tedious, and methods have been developed to infer types for most expressions. The type system used in DILS is similar to that used by ML, as described in Milner's theory of type polymorphism [Mil78]; a unification-based type inferencer is provided so that only operators and top-level labels need to have their types declared, the former to give the type-inferencer something to work with and the latter due to technical difficulties arising from trying to infer types for mutually-recursive polymorphic functions. ML style polymorphism is allowed for both operators and top-level labels. Internal labels and lambdas cannot be polymorphic. The type inferencer is based on traditional techniques used in ML and is not discussed in this dissertation; the interested reader is referred to Cardelli's basic treatment [Car85].

Although DILS does not define the semantics of a given type, it does fix the type rules necessary to infer the type of any expression from the type declarations for operators and top-level labels. These rules are as follows:

Constants

Strings have the primitive type `string`. Integers have the primitive type `integer`. An operator may have any type; the type is included in the operator definition.

Applications

In an application `(`*func arg...*`)`, the *func* must have type `(fn` *conv return-type arg-type...*`)` where *conv* is a *calling convention* and the *arg-type*s are the types of the corresponding *arg*s. *return-type* denotes the type of the value returned by the application. `fn` is a primitive type used to construct functional types. Calling conventions are types used to pass information throughout the compilation process on decisions made regarding the manner in which applications of a given function will be implemented. Details on particular calling conventions used in Dora are presented in Chapter 6 on the UOPT prototype.

Lambda expressions

Lambda expressions have type `(fn` *conv return-type parm-type*...`)` where *conv* denotes the expected calling convention to be used to implement the lambda, *return-type* gives the type of the value returned when the lambda is applied, and the *parm-type*s give the type and number of parameters that must be supplied.

Variables

Variables, in and of themselves, are not expressions, but must be used either in the value form `^x` or the cell form `&x`. The type of the expression `^x` is inferred from the use of the lambda containing x; the type of the expression `&x` is `(cell` t`)`, where t is the type of `^x`.

Labels

The type of a `labels` expression is the same as the type of the corresponding body. The type of a label is the type of the corresponding lambda (explicitly defined in the `def-label` form, or inferred for labels defined in a DILS `labels` expression).

Continuations

In a continuation capture `(c-cap` *func*`)`, *func* must have type `(fn` *conv return-type* `(cont` *return-type*`))` for some calling convention *conv* and type *return-type*; the capture expression has type *return-type*.

In a continuation application `(c-app` *cont val*`)`, *cont* must have type `(cont` *val-type*`)` where *val-type* is the type of *val*. The type of the continuation application is `void` (since continuation applications throw away their current continuation, they should only be used in contexts where no value is expected; ignoring a return value is expressed, by convention, with the type `void`).

2.6 Why DILS?

This chapter opened by noting that an intermediate language for Dora should be general, in order to allow handling a wide variety of languages and machines, and semantically rich and detailed, to enable optimization code to be widely applicable and independent of IL independent assumptions. Intermediate languages exist at both ends of this scale; the MUG2 system [Wil81] provides a general tree data structure that is given all its meaning by the individual compiler writer, while many production compilers use a representation tailored specifically to the particular source language and target machine involved.

While our two goals may seem to be in conflict, DILS succeeds in meeting them both. DILS is linguistically based on the lambda calculus with imperative extensions. The basic concepts of the lambda calculus (lambda expressions, applications, and variables), extended with the concept of a store, form the core of the Scheme programming language, which was proposed by Steele as an excellent intermediate language for all languages [Ste76].

Compilation methods invented since Steele expressed his argument, particularly the closure analysis methods of Kranz et al. [KKR86] and control-flow analysis methods of Olin Shivers [Shi88], have made it possible to implement control-flow expressed with lambda expressions as efficiently as control-flow expressed with GOTOs. Kelsey has validated this work by building a Pascal compiler using a lambda-calculus based IL that produces efficient machine code [Kel89]. The construction of a UOPT prototype using DILS, performing the same optimizations as one of the most recent and complete optimization suites available for traditional basic-block oriented ILs, strengthens Steele's claim.

DILS contains two further extensions to the core Scheme concepts: the continuation capture and types. With continuation captures, the compiler writer can model non-local control flow in a natural way without resorting to the clumsy continuation-passing-style convention. Felleisen [Fel87] argues strongly that the continuation capture construct is an appropriate imperative extension to the lambda calculus, as useful as the store, and shows how formal reasoning with continuation captures can be performed. Typing is an important extension to easily handle the preponderance of static information that would otherwise be represented explicitly in statically typed languages. With these two extensions, DILS is an excellent IL for both imperative and functional programming languages.

Chapter 3

Tree pattern matching

An old criticism of tree-oriented intermediate languages is that traditional tuples are easier to manipulate. Dora provides extensive tree-oriented analysis and transformation facilities in answer to this criticism. Both the analysis language and the transformation language are based heavily on pattern matching. Since the pattern language used in Dora makes contributions to the widely-applicable pattern matching problem, patterns are first discussed in this chapter in a relatively independent setting before interactions with Dora are covered.

The *tree pattern matching problem* — given a fixed set of tree patterns and an arbitrary input tree, attribute each node of the tree with the set of all matching patterns — has many useful applications, particularly within the realm of compilation. In many application areas, including the uses in Sal and Tess, the efficiency of the pattern matcher is important. There are two important factors impacting the efficiency of pattern matching in these systems. First of all, it is important to take into account that matches are wanted at all the nodes of a tree. While investigating a given node, the matcher should take into account not only what patterns match at that node, but what effect this node has on pattern matches at nearby nodes. Secondly, there may be a large number of patterns being matched simultaneously; therefore, a matcher that can match a potentially unbounded number of patterns at once will have a large advantage over a matcher that is slowed down by additional patterns.

Hoffman and O'Donnell have presented efficient automata-based algorithms for solving the pattern matching problem with relatively simple pattern languages, in which a pattern is simply a tree with (possibly typed) wildcards at some leaves [HoO82]. David Chase made a significant advance in the practicality of these systems by providing a large reduction in the time and space requirements of the automaton construction algorithm [Cha87]. But the simple pattern language used by Hoffman, O'Donnell, and Chase is less than ideal for many applications. Typical extensions involve dealing with variable-arity operators, including "splice wildcards" that match zero or more children of a node (instead of exactly one), and including some escape to base-language predicates within the patterns. Pattern matching systems using these extensions can be very powerful, very complicated, and very slow.

The automata-based algorithms have the important property that the speed of

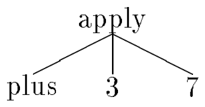
Formal definition	Pictorial	Textual
$\begin{aligned}\text{Nodes} &= \{a,b,c,d\}\\ L &= \{(a,\text{apply}),(b,\text{operator}),\\ &\quad\;(c,\text{integer}),(d,\text{integer})\}\\ C &= \{(a,(b,c,d)),(b,()),(c,()),(d,())\}\\ D &= \{(b,\text{plus}),(c,3),(d,7)\}\end{aligned}$		`[apply plus 3 7]`

Figure 3.1: Three representations of trees

the matcher is independent of the number of patterns to be matched. In applications with a large number of patterns, this property can be critical to the successful use of pattern matching, and thus these applications have been forced to make do with the simple pattern matching language.

In this chapter, two extensions to the traditional pattern-matching constructs — horizontal and vertical iterators — are introduced. Horizontal iterators enhance the pattern matcher's ability to deal with variable-arity operators. Vertical iterators allow a pattern to specify a repetitive "backbone" between the root of the pattern and a leaf. The notational enhancements of the constructs can be handled without slowing down the pattern matcher; an automated reduction to the simple pattern matching language allows the automata-based solutions to be used for the enhanced language.

Most of the examples in this chapter will come from abstract syntax trees, rather than DILS trees, as the former are more compact. Section 3.3 describes an alternate notation for more compact specification of DILS patterns that will be used in future chapters.

3.1 What's a tree?

This section briefly lays out the terminology and notation we will be using. Figure 3.1 contrasts the formal definition with a pictorial representation and the text representation used in Dora.

A *tree* over a finite set $\mathcal{L}$ is a set of nodes with a labeling function L, children function C, and data function D. The labeling function assigns to each node a member of $\mathcal{L}$, called a *label*. Labels come in two varieties, *interior* labels and *data* labels; nodes are correspondingly called interior nodes or data nodes depending on the variety of their label. A data label d has an associated set of constants $\mathcal{D}(d)$ called the *domain*.

The children function C assigns to each internal node a (possibly empty) list of other nodes, its children; C assigns the empty list to each data node. The data function D assigns to each data node n a constant in $\mathcal{D}(L(n))$. Data labels make it possible to incorporate infinite constant sets (integers, for example) in the tree while keeping the set of labels finite; a finite number of labels is an important constraint in building fast pattern matchers. For any two different labels d_1 and d_2 in the same tree, $\mathcal{D}(d_1) \cap \mathcal{D}(d_2) = \emptyset$; thus, for data nodes, one can deduce $L(n)$ from $D(n)$.

For each node x, there is at most one node y, the *parent* of x, such that x is a child of y. Exactly one node, the *root*, has no parent. Nodes with no children are called *leaves*. Two trees are equal if they are isomorphic with respect to the labeling, children, and data functions; i.e., nodes have no interesting properties other than those expressed through these functions.

3.1.1 Textual representation

A data node n is textually represented by its constant $D(n)$, which determines the node label since domains do not intersect. I will restrict my examples to three different constant domains: atoms (represented with arbitrary alphanumeric strings, such as `plus`), strings enclosed in double quotes (for example, `"a string"`), and integers.

An interior node is represented by a list enclosed in square brackets; the first element of the list is the label of the node, and the rest of the elements form the children list. Labels are represented with alphanumeric strings, and are differentiated from atoms by their position as first member of the list. Thus, `[apply plus 1 2]` is a tree with a root node labeled `apply`, with three data nodes as children.

3.2 Patterns

A *pattern* is a character string used to specify a subset of trees, those trees the pattern *matches*. Typical pattern matching systems can be divided into two different camps. The first camp uses very simple patterns, sometimes extended with a syntactic type facility, and performs heavy preprocessing on sets of patterns to build efficient pattern matchers. A survey of these systems can be found in Pelegrí's dissertation [Pel88]. The second camp devises complicated pattern languages, which are essentially interpreted at matching time. A few pattern matchers in this category are discussed in the pattern-matching chapter of Barr and Feigenbaum's *Handbook of Artificial Intelligence* [BaF82].

Although the traditional patterns handled by tree automata are not as descriptive as some of the more powerful interpreted pattern systems, they have a strong advantage in being able to match a large number of patterns simultaneously. An arbitrary (finite) set of patterns can be preprocessed to produce a pattern matcher that, in a single walk over the tree, decorates each node with the set of all patterns that match at the node. The speed of this tree walk is independent of the number of patterns to be matched, an important consideration in compilation applications where many different patterns are used in an extensive case-analysis.

The traditional automata-based pattern-matching language consists of simple trees extended with typed wildcards and variables. In many situations, it is desirable to express some kind of horizontal and/or vertical repetition within a pattern. Two extensions to the traditional pattern matching language, regular expression child lists and vertical iterators, can express this repetition without destroying the automata-based implementation of the matcher. This section describes the traditional pattern based constructs and the extensions; Section 3.5 describes the implementation of the matcher.

3.2.1 Simple wildcard patterns

The simplest patterns are formed from the textual representation of some tree in which zero or more of the node representations are replaced by *wildcards*. A wildcard is a string of the form `?` or `?.`*type*, where *type* is a regular set of trees as specified below.

A pattern p matches a tree t if

1. p is the textual representation of t, or
2. p is `?`, or
3. p is `?.`*type* and t is a member of *type*, or
4. p is of the form `[`l $c_1 \ldots c_n$`]`, $l = L(t)$, t has n children, and each pattern c_i matches the i'th child of t.

For example purposes, we shall used the type `int`, which is the set of integer data nodes. Thus, the pattern `[plus 1 2]` matches the tree `[plus 1 2]`, `[plus ? ?]` matches all nodes labeled `plus` with two children, and `[plus ?.int ?.int]` matches all nodes labeled `plus` with two integer constants as children.

Only *regular* tree sets may be used as types. Regular tree sets are an analogue of regular sets of strings, those strings recognized by a finite-state automata; regular tree sets are those tree sets recognized by a finite-state bottom-up tree automata, a machine that produces a new state by performing a table lookup based on the operator of a node and the states of its children [Tha73]. A type, or regular set, is specified in one of two ways. First, there is a type associated with each of the data operators; in our examples, these types are `int` and `str`, for integer and string constants respectively. Secondly, a type may be specified by giving a list of patterns:

```
(define-pattern-type type
  pattern ...)
```

defining *type* to be the set of all trees matched by one or more of the *pattern*s. For example,

```
(define-pattern-type simple-arith-tree
  [plus ? ?]
  [minus ? ?])
```

defines `simple-arith-tree` to be the set of all trees with a *plus* or *minus* operator at the root; `?.simple-arith-tree` would then be a pattern matching all of these trees. The patterns listed may have recursive references to the type being defined; for example,

```
(define-pattern-type sat-2
  [plus ?.sat-2 ?.sat-2]
  [minus ?.sat-2 ?.sat-2]
  ?.int)
```

describes the set of all trees composed solely of `plus` and `minus` nodes, with integer constants at the leaves.

3.2.2 Horizontal iterators

Traditionally, tree pattern matchers have used only the kinds of patterns described in the previous sections, namely trees extended with wildcards. It is also common to restrict the labels to having a fixed *arity*; all nodes with a given label are restricted to have the same number of children. This restriction is inconvenient in many applications; in particular, abstract syntax trees in Pan [BGV90] can have an arbitrary number of children (to handle lists or sequences naturally) and DILS applications can have an unbounded number of children. Since the restriction only provides a small constant factor of improvement in algorithms dealing with pattern matching, it is not used in Dora.

Horizontal iterators provide a way to match a sublist of a node's children without restricting the size of the sublist. Within a child list, any sublist may be enclosed in braces. Loosely speaking, the braces can be read as "insert zero or more copies of our contents into the children list". For example, the pattern `[plus {1}]` matches the nodes `[plus]`, `[plus 1]`, `[plus 1 1]`, and so on. Horizontal iterators can be nested; thus, `[plus {1 {2 3}}]` matches `plus`-labeled nodes with either zero children, or with a 1 followed by any number of 1's and/or 2-3 pairs.

Horizontal iterators can be formally defined using concepts similar to string regular expressions, using patterns as the alphabet. The brace operators are taken to be a combination grouping and Kleene star operator, that is, `{...}` is equivalent to `(...)*`. Rule 4 in the definition of matching is replaced with

4′. p is of the form `[`l c`]` where c is a regular expression (without alternation) using patterns as the alphabet, $l = L(t)$, and there exists a string of patterns c_1 through c_n generated by c such that t has n children, and each pattern c_i matches the i'th child of t.

This concept can be extended by providing a notation for alternation, allowing the full power of regular expressions to be used in the child list. Dora does not provide such a notation since there has been little motivation for its use and no clear choice on how to deal with pattern variables in such a construct. The implementation described in section 3.5 is easily extended to deal with alternation, if a reasonable notation is discovered.

3.2.3 Vertical iterators

Horizontal iterators provide the power to match zero or more children of a node. On occasion, it is desirable to describe trees with zero or more identical "wrappers" built up on some base node. This is best described with an example: suppose we have a binary `plus`-labeled node, and `plus`-trees are made left-heavy by convention. We might want to identify all trees with zero or more additions to a constant, such as `3`, `[plus 3 4]`, `[plus [plus 3 4] [var "x"]]`, and so on.

Such trees have some base pattern, `?.int` in our example, and a wrapper that is to be placed around the base, such as `[plus ... ?]`. We can match such patterns with a *vertical iterator*, specified by prefixing the wrapper with a `$` and enclosing the base pattern in dots, in its appropriate place within the wrapper. Our `plus`-tree example is matched by the pattern `$[plus ... ?.int ... ?]`.

Vertical iterators are matched using the following rule:

5. p is a vertical iterator of the form `$[`*l left-wrapper ... base ... right-wrapper*`]` where *base* is a pattern,
   base is not in a horizontal iterator in the pattern `[`*l left-wrapper base right-wrapper*`]`,
   and either *base* or the pattern `[`*l left-wrapper p right-wrapper*`]` matches t.

3.2.4 Variables

Having discovered that a particular tree is matched by a given pattern, it is often desirable to refer to portions of the tree matched by wildcards in the pattern. This can be done by using *variables*. A variable is a named wildcard, denoted `?`*name* (with an optional `.`*type* following).

A pattern in which each variable name occurs at most once is said to be *linear*. The usual interpretation of a non-linear pattern, i.e., a pattern with a repeated variable, is that the pattern only matches trees in which the subtrees matched by the repeated variables are identical.

The most practical current implementations of non-linear patterns perform a linear match, followed by a test to see if the associated subtrees are equal; this is easily done within Dora by using a linear pattern with an explicit equality test. Furthermore, combining non-linear patterns with iterators can easily lead to patterns that require exponential time for matching. For these reasons, Dora only accepts linear patterns.

When a variable is contained within a horizontal and/or vertical iterator, it conceptually "matches" several (or perhaps zero) subtrees. For unnested iterators, the solution is relatively straightforward: the variable is bound to a list of the matched subtrees. For example, `[plus { ?arg }]` matches `[plus 1 2 3]` with `arg` bound to the list `(1 2 3)`, and `$[plus ... ?x.int ... ?y]` matches `[plus [plus 3 4] [var "x"]]` with `x` bound to `3` and `y` bound to `([var "x"] 4)`. The list is ordered left-to-right for horizontal iterators, and top-down for vertical iterators.

For nested iterators, the situation is more complex. Consider a match of the pattern `[plus {"x" {?i.int} }]` with the tree `[plus "x" 1 2 "x" 3 4 "x" "x" 5]`. One possibility would be to bind `i` to a list of the elements matched, that is, `(1 2 3 4 5)`. But doing so loses some of the structural information implied by the pattern, namely that there are subgroups of integers separated by `"x"`'s. Dora maintains this structure by binding `i` to the list `((1 2) (3 4) () (5))`. This example is generalized as follows. For a variable contained directly within the iterator (that is, not contained within some sub-iterator), the iterator binds the variable to a list of the values (left-right or top-down) matched by the wildcard. For a variable contained within a sub-iterator, the iterator binds the variable to a list of the values (left-right/top-down) produced to the variable by the copies of the sub-iterator. The primary advantage of this method of binding variables in iterators is that it allows pattern instantiation (described in the following section) to function properly with nested iterators.

3.2.5 Pattern instantiation

Patterns are not generally created dynamically; instead, a group of patterns is statically analyzed to build an efficient matcher. But the ability to specify simple sub-patterns (that is, sub-patterns matching a single tree) dynamically is frequently useful; for example, having found a variable bound in a special lambda expression, it might be desirable to find all assignments to the variable.

Within a pattern, a previously-bound variable prefixed with an exclamation point can be used to match the tree bound to the variable. These patterns are called *instantia-*

tions, matched with the rule:

6. p is the instantiation `!`v, and v is bound to t.

In our example, the lambda variable would be bound to a Lisp variable, say `var`, and all assignments could be found by the pattern `[assign [var-address !var] ?val]`.

Instantiations may appear within iterators. The instantiation variable must be bound to a list. The (dynamic) length of this list determines the number of copies made by the iterator; one element of the list is used in each copy, in order left to right or top-down. Thus, if `arg` is bound to the list `(1 2 3)`, then `[plus { !arg }]` matches the tree `[plus 1 2 3]`. If there are several `!` expressions within an iterator, each resulting list must have an identical length.

When appearing within nested iterators, instantiation variables must be bound to appropriately nested lists. The general rule for dealing with instantiations is

6′. p contains instantiation variables $v_1, \ldots, v_n$ and the pattern p' formed by replacing each $!v_i$ in p with $?v_i'$ matches t binding each v_i' to v_i.

This points out the dual relationship between `?` and `!`. In the nested iterator example, the pattern `[plus {"x" {?i.int} }]` matches the tree t = `[plus "x" 1 2 "x" 3 4 "x" "x" 5]` by binding `i` to the list `((1 2) (3 4) () (5))`. With `i` bound to this nested list, the pattern `[plus {"x" {!i} }]` matches t and no other trees.

A pattern with no wildcards or variables matches at most one tree. (It may not match any trees if there are nested instantiation variables enforcing differing structures on the tree). *Pattern instantiation* is the act of creating this tree. Pattern instantiation plays a basic role in the Tess tree rewriting system of Dora.

3.3 Special notation for DILS

For some tree languages it can be convenient to have a shorthand notation for certain recurring patterns. For example, pattern matching nested applications in DILS code quickly becomes tedious; matching the DILS expression `(plus 1 (plus 2 (plus 3 4)))` requires the overly cumbersome pattern `[app plus 1 [app plus 2 [app plus 3 4]]]`. Hooks have been provided in the pattern matcher to allow simple extensions to the pattern language that map directly onto the primitive patterns. In particular, extensions have been provided for writing patterns that mimic the DILS external representation. DILS trees use a small set of node labels: `app`, `c-app`, `c-cap`, `lm`, `labels`, `label-def`, `^`, `&`, and data labels for integers, strings, and DILS atoms. Nodes with these labels are textually represented as follows:

- A list enclosed in parentheses represents an `app`-node with the list elements as children, unless the first element of the list is a node label. For example, `(plus ?x ?y)` is an alternative notation for `[app plus ?x ?y]`.

- The node labels `c-app`, `c-cap`, `^`, and `&` can be used with parentheses in place of square brackets.

- `(lm ({?vars}) ?body)` is equivalent to `[lm ?body {?vars}]`; in an `lm`-labeled node, the first child is the body and the rest are the list of variables.

- `(labels ({(?labs ?lams}) ?body)` is equivalent to `[labels ?body {[label-def ?labs ?lams]}]`; in an `labels`-labeled node, the first child is the body and the rest form the list of label definitions.

The reader is advised to become familiar with this notation and refer to this section when necessary, as the DILS pattern notation is used in most of the remaining examples in the dissertation.

Pattern instantiation, using this DILS shorthand, is a convenient way to construct DILS trees both for testing purposes and within phases of a compiler. The Lisp macro `build-dils` is provided to perform this function; it takes one argument, a DILS pattern, and instantiates it. For example, with the variable `args` bound to the list `(1 2)`, the form `(build-dils (plus {!args}))` will return the DILS node `[app plus 1 2]`.

3.4 The `tree-match-case` form

One of the main strengths of Dora's pattern language is that an automaton can be constructed to find all the matches at each node of a tree in a single tree walk. By far the most common use of pattern matching is made in Sal attributes and Tess transformations, which have their own special forms for making use of pattern matching. But it is also useful to have more direct access to the pattern matcher, for use during debugging, or for use of the pattern matcher outside of the context of Dora. The `tree-match-case` form encapsulates the solution to the tree pattern matching problem.

`tree-match-case` is a macro, which can be used by arbitrary Lisp code, that is used to select an action based on the local context of a tree node. Uses of `tree-match-case` take the form

`(tree-match-case (`*var node-form*`)` *optional-keywords*
  `(`*pattern Lisp-forms*`) ...)`

The *node-form* is a Lisp form evaluating to a tree node *n*. `tree-match-case` walks the tree rooted at *n*, in an order specified by the optional keywords. At each sub-tree visited in the walk, the (*pattern Lisp-forms*) clauses are scanned until one whose pattern matches the sub-tree is found. The Lisp-forms of this clause are then evaluated, with *var* bound to the sub-tree and the variables in the pattern bound to the matching nodes. As a simple example, one could print each of the variable occurrences in a subtree `the-tree` thus:

```
(tree-match-case (node the-tree)
  ((^ ?var)
   (print var))
  ((& ?var)
   (print var)))
```

3.4.1 Multiple matches

A pattern with variables in iterators could conceivably match a single node with many different bindings of the pattern variables. For example, the pattern `[foo {?x} {?y}]` could match the node `[foo 1]` with `x` bound to `(1)` and `y` bound to `()`, or vice versa. If a single pattern matches a given node multiple times, the associated Lisp forms are evaluated multiple times, once for each set of variable bindings making a match. The order of evaluation is undefined. For example, the form

```
(tree-match-case (node (build-dils (plus 1 2)))
  ((plus {?initial-args} ?arg {?rest-args})
   (print-node arg)))
```

might yield the output `2 1`.

3.4.2 Controlling the tree-walk and testing all patterns

`tree-match-case` has parameters that may be used to control the order in which the tree is walked, or to request that all patterns be tested. To control the tree-walk, at most one of the keywords `:top-down` or `:bottom-up` may be specified, and at most one of the keywords `:right-left` or `:left-right`. Inclusion of these keywords constrains the order of the sub-trees visited in the obvious way.

By default, `tree-match-case` stops scanning clauses for a given sub-tree when it has found the first matching pattern clause. If the `:test-all-patterns` keyword is included, each clause is tested regardless of the success or failure of earlier clauses.

3.4.3 Naming interior nodes

Within a `tree-match-case` form (and in other contexts where pattern variables get bound) it is sometimes desirable to name interior nodes of a tree matched by a pattern. This can be done by prefixing the desired node in the pattern with `?`*var*`=`; for example

```
(tree-match-case (node (build-dils (plus (plus 1 2) 3)))
  ((plus ?arg1=(plus ? ?) ?)
   (print arg1)))
```

will print `(plus 1 2)`.

3.5 The implementation of `tree-match-case`

Chapters 4 and 5 describe how this pattern language is used within Dora to assist in analyzing and transforming DILS code. The remainder of this chapter describes, at a high level, the implementation of the `tree-match-case` macro. The `tree-match-case` problem can be broken into two parts; first, annotate each node of the target tree with the set of all patterns matching at that node, and secondly walk the tree, picking the appropriate matching pattern(s), binding their variables, and executing their associated Lisp forms. The first part is simply the tree pattern matching problem.

3.5.1 Tree pattern matching with iterators

There is an extended literature on efficient solutions to the tree pattern matching problem for simple patterns with wildcards; Pelegrí gives a suitable survey and implementation details [Pel88]. I will restrict this discussion to showing how to deal with the extended patterns by reducing them to equivalent simple patterns. The reductions make heavy use of the type system allowed by the regular tree sets, and are similar to the implementation of regular string expressions with regular grammars.

Vertical iterators

Vertical iterators are trivial to implement with typed wildcards. Consider a general vertical iterator `$[`*left-wrapper ... base ... right-wrapper*`]`. This pattern is equivalent to `?.t`, where the type `t` is defined as

`(define-pattern-type t`
  base
  `[`*left-wrapper* `?.t` *right-wrapper*`])`.

The proof is trivial from the definition of pattern types and rule 5.

Variable-arity nodes

The first step in dealing with horizontal iterators is forcing all operators to have a fixed arity. This is done by introducing two new operators, $\mathcal{S}$ and $\mathcal{E}$, as special "start" and "end" brackets, and then forcing all nodes to have arity 2 by converting children lists into pairs of the beginning of the list and the next element: the node `[`l c_1 ... c_n`]` is converted to the node `[`l`[`l $\cdots$ `[`l `[`$\mathcal{S}$`]` c_1`]` $\cdots$ c_n`]` `[`$\mathcal{E}$`]]`. This conversion is applied both to the pattern set and to the subject tree; it is applied textually to the pattern set by considering sublists grouped in the horizontal iterator brackets as single children. It is easy to show that a solution for the converted problem is equivalent to a solution for the original. The implementation can do the transformation conceptually without actually modifying the trees.

Horizontal iterators

A horizontal iterator must match its subpatterns zero or more times against the target node. After the variable-arity to binary conversion, the iterator must be the right child of some binary operator, i.e., it must be in the form `[`l p `{`$c_1 \ldots c_n$ `}]`. In order for this portion of the pattern to match some partial tree `[`l t_1 t_2`]`, we need to "insert" zero or more copies of c_1 through c_n into the pattern and have the resulting pattern match. But this form of vertical recursion is easy to do with typed patterns: we can replace the pattern `[`l p `{`$c_1 \ldots c_n$ `}]` with the pattern `?.t`, where the type `t` is defined as

`(define-pattern-type t`
  p
  `[`$l\cdots$ `[`l `?.t` c_1`]` $\cdots$ c_n `])`.

The wildcard `?.t` either allows p to match directly, handling the case with zero insertions of the iterator's children, or inserts one copy and allows a recursive match.

3.5.2 Binding variables

The solution to the tree pattern matching problem efficiently annotates each node of the target tree with the set of all matching (sub)patterns. But this is not a total solution for the implementation of `tree-match-case`, as we still need to find appropriate variable bindings. Finding these bindings can be done with a simple top-down walk of the target tree, with actions at each node driven by corresponding nodes of the pattern. An interpretive algorithm, taking both the matching pattern and the target tree as input and returning a set of environments, is presented in Figure 3.2.

The binding algorithm uses the binary representation of trees, with the tree-type representation of iterators; the *wrapper* and *base* of an iterator are, respectively, the subpatterns in the iterator type that do and do not recursively include the iterator type. In the figure, `()` is used to denote the empty environment and angle brackets enclose sequences. Variables give rise to new bindings in the environment, either directly to the corresponding tree, or to a sequence containing the tree if the variable occurs within the scope of an iterator. Iterator patterns can give rise to more than one environment in the *bindings* set. If an iterator base matches at a tree (this match is done with a simple test of the state computed by the global matching walk), then the variables directly nested within the iterator must be bound to the empty sequence, to handle the case of zero iterations. The cross-product operator for `binary-node`s requires special handling; it denotes that, for each pair of environments ρ_l and ρ_r from the left and right child, ρ_l should be extended by ρ_r with the resulting environment ρ added to *bindings*. Environment extension involves simply adding the new bindings when outside the scope of any iterators; when inside the scope of an iterator, a variable bound to a sequence both in ρ_l and ρ_r should be bound to the concatenation of the sequences in ρ.

The algorithm as presented takes the pattern as an argument, and computes sets of bindings. Standard algorithmic techniques are used within Dora to compile a particular binding algorithm for each particular pattern, optimized to shortcut portions of the walk that produce no bindings and to use the bindings as they are created, rather than building a large set in memory.

```
procedure bind_pat_vars(pattern, tree)
  var bindings : set of environment;
  case pattern_type(pattern) of
    variable:
      if in the scope of an iterator then
        bindings := { (variable(pattern) $\mapsto$ $\langle$tree$\rangle$) };
      else
        bindings := { (variable(pattern) $\mapsto$ tree) };
    wildcard:
      bindings := { () };
    horizontal_iterator:
      bindings := { };
      if horizontal_base(pattern) matches tree then
        add bind_pat_vars(horizontal_base(pattern),tree) to bindings;
        extend environments in bindings by (var $\mapsto$ $\langle\rangle$) for each
          var directly nested in pattern;
      end if;
      if horizontal_wrapper(pattern) matches tree then
        add bind_pat_vars(horizontal_wrapper(pattern),tree) to bindings;
      end if;
    vertical_iterator:
      bindings := { };
      if vertical_base(pattern) matches tree then
        add bind_pat_vars(vertical_base(pattern),tree) to bindings;
        extend environments in bindings by (var $\mapsto$ $\langle\rangle$) for each
          var directly nested in pattern;
      end if;
      if vertical_wrapper(pattern) matches tree then
        add bind_pat_vars(vertical_wrapper(pattern),tree) to bindings;
      end if;
    leaf_node:
      bindings := { () };
    binary_node:
      bindings := bind_pat_vars(left_child(pattern),left_child(tree)) $\times$
                  bind_pat_vars(right_child(pattern),right_child(tree));
  end case;
  return bindings;
end procedure
```

Figure 3.2: Interpretive binding algorithm

Chapter 4

Sal

Semantic analysis plays an important role in optimization. The attribute model, in which nodes are labeled with attributes that summarize the results of some analysis, has been used quite successfully in the past. Sal continues this tradition; the end goal of a Sal analysis is a set of attribute values for the nodes of a tree. These attribute values are accessed by Lisp-callable functions, which perform the analysis when necessary (for example, if the tree has been modified since the attribute value was last computed).

The most common analysis method making use of the attribute model is the *attribute grammar*. An attribute grammar extends the traditional BNF description of the concrete or abstract grammar of a language with *attribute equations*, which define certain attributes of parse-tree nodes in terms of attributes of nearby nodes. Attribute grammar systems such as Linguist [Far89] and GAG [KHZ82] have been used for many compilation tasks from simple type checking through code generation. It is assumed that the reader has been previously exposed to the concept of attribute grammars. Background material may be found in a survey by Deransart, Jourdan, and Lorho [DJL88].

Among the major strengths of attribute grammars as analysis tools are the ability to give a functional description of node attributes and the built-in local context given by the grammar. But attribute grammars have distinct disadvantages when used for large real-world systems. Primary among these are the monolithic description produced by coupling the attribute equations with the grammar and the inflexibility of the grammar itself. Sal, the semantic attribute language used in Dora, enhances the strengths of the attribute grammar while removing the tight coupling with the syntactic description of the language.

Sal is used in Dora for many different kinds of analysis. Although data-flow analysis consumes much of the run-time of many optimizers, it forms a relatively small amount of the code that must be written; for example, in the prototype of UOPT, an optimizer that is designed to move as much of the analysis as possible into the data-flow portion of the optimizer, only one fourth of the Sal attributes are of the data-flow nature. The remaining three quarters compute the control-flow graph, set up local attributes for the global data-flow problems, and solve the multitude of small but necessary problems inherent in compilation. This is not to say that data-flow is unimportant, as it does dominate the run-time of the resulting optimizer; Douglas Grundman is investigating a special-purpose data-flow tool designed both for ease of specification and for efficiency [Gru90].

In this chapter, we begin by looking at the advantages and disadvantages of the attribute grammar as a program analysis tool, and then show how Sal enhances the advantages while solving some of the problems.

4.1 Program analysis using attribute grammars

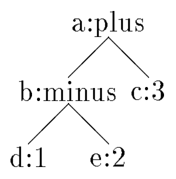
Attribute grammars were originally proposed as a method for describing language semantics [Knu68]. Since their inception, they have been used throughout the compilation process. The attribute grammar is now viewed as a general tree-attribution tool, suitable for optimization analysis [LMW88], circular data flow analysis [Far86], and even tree-transformations [VSK89].

Attribute grammars have many excellent properties as analysis tools. The grammar itself provides a complete description of the language under consideration, performing double duty as both the syntactic description and a complete framework for the semantic description. An attribute grammar is a reasonably *declarative* specification; the grammar writer does not implement a tree walk, and the language for specifying attribute values is side-effect free. (It must be, since the order of attribute evaluation is unspecified.) Much work has been invested in producing efficient and incremental attribute grammar evaluators. The grammar syntactically divides up the language, thus providing some simple but powerful built-in case analysis. Finally, most attribute grammars also provide a convenient naming system for referring to the children of a given node, using names present in the grammar production.

Unfortunately, attribute grammars have some disadvantages as well. An attribute grammar forms a monolithic description: the definitions for all attributes are spread throughout a single large grammar. Adding a new attribute may only require adding definitions to three or four productions, but those productions may be spread across hundreds of pages. Trying to modify a particular attribute involves rummaging through the grammar to find all the definitions involving that attribute. Although the grammar provides syntactic modularity, corresponding semantic modularity is absent.

The attribute grammar simultaneously describes the syntax and semantics of the language. This simultaneous definition is convenient for some purposes, but it leads to difficulty when modifying the description. Trying to modify a single grammar production involves understanding not only the syntactic relationships of the particular terminals and non-terminals involved, but also the definitions of all of the attributes described in the production. There is a tight inter-reliance of the syntactic and semantic description, which becomes untenable in a large and changing description. In addition to this syntactic/semantic interaction, different semantic attributes are also tightly-coupled through the grammar. If a particular attribute needs a slightly different case analysis from other attributes, it can only be obtained through grammar modification, to the detriment of the other attributes, the grammar's maintainability, or both.

The problem of the monolithic description is well known, and several researchers have proposed different solutions. All involve factoring the description in some way, some by copying the grammar (or an abstracted grammar) for each attribute [NoP88], others by using some form of pattern matching on productions to automatically create rules for many



$$\begin{aligned}
\mathrm{Val}_a &= \mathrm{Val}_b + \mathrm{Val}_c \\
\mathrm{Val}_b &= \mathrm{Val}_d - \mathrm{Val}_e \\
\mathrm{Val}_c &= 3 \\
\mathrm{Val}_d &= 1 \\
\mathrm{Val}_e &= 2
\end{aligned}$$

Figure 4.1: Tree and corresponding set of attribute equations for calculator example

different productions with one specification [DuC88]. These solutions all can be viewed as "preprocessors"; logically, if not explicitly, they create a traditional monolithic attribute grammar from a factored description. As such, they maintain the disadvantages of the underlying grammar — the inter-reliance of semantic and syntactic descriptions, and the fixed set of production rules to perform case-analysis for different attributes — without maintaining the grammar's advantage of a single complete description.

4.2 The role of the grammar in AG systems

As seen in the previous section, some of the disadvantages of attribute grammars are inherent in any system based on an underlying grammar. In order to escape these disadvantages, which are fatal to any system aiming at ease of modification, the reliance on the grammar has to be removed. To remove this reliance, the contribution of the grammar needs to be understood. Any attribute grammar system can be broken into three subsystems: the grammar language and association of attribute equations with grammar productions, the semantic equation language, and the evaluation engine that solves the set of attribute equations associated with a particular tree.

The grammar itself plays the role of specifying the set of attribute equations associated with a given input tree. Consider, for example, a simple production from an attribute grammar for a calculator language:

```
expr ::= integer { expr.val = literal_val(integer); }
     |   expr PLUS expr { expr_1.val = expr_2.val + expr_3.val; }
     |   expr MINUS expr { expr_1.val = expr_2.val - expr_3.val; }
```

Figure 4.1 shows a simple tree and the set of attribute equations described by this production for the given tree. The grammar specifies the set of equations, but its interaction with the attribute equation language is limited to the method of specifying attribute values as arguments to the language.

The second subsystem, the attribute equation language, is independent of the grammar formalism, except for an interface for referencing nodes that appear in the grammar productions. The language must be functional, as there is no specification of the order in which various expressions might be evaluated, but otherwise is relatively unrestricted.

The third subsystem, the evaluation engine, is more intimately connected with the grammar itself: many different evaluation strategies have been proposed for attribute grammars, varying in power but generally quite efficient. Notable advances include Reps's incremental evaluator [Rep84] and evaluators that handle circular dependencies among attribute equations, such as those of Farrow or Walz and Johnson [Far86, WaJ88]. These evaluators generally analyze the grammar to determine possible dependencies among the attribute equations, to aid in incrementality and/or efficiency.

Having outlined the role of the grammar, we are in a better position to see what needs to be replaced. Many of the important concepts of the attribute grammar, such as the functional description of attributes and an automatically generated evaluation engine to handle attribute dependencies, can be kept; we simply need a different specification language for the attribute equations. The specification language should maintain the good case-analysis and node-naming properties of the grammar-based specification. These properties of the grammar can be handled easily by pattern-matching.

4.3 Describing a set of equations in Sal

In an attribute grammar, the set of attribute equations is specified by listing, for each production in the grammar, a prototypical set of equations for attributes of the node instances created by that production. In Sal, this role of the production is taken over by tree pattern matching. A production in a context-free grammar corresponds to any node in the parse tree with the left hand side as the operator and the right hand side giving the types of the children; patterns matching these constructs are trivial to write. For example, the earlier calculator example productions could be mimicked by a description such as

```
[expr ?x.integer]               { expr.val = literal_val(x); }
[expr ?x1.expr PLUS ?x2.expr]   { expr.val = x1.val + x2.val; }
[expr ?x1.expr MINUS ?x2.expr]  { expr.val = x1.val - x2.val; }
```

But there is no need to restrict oneself to mimicking the original grammar. The patterns, and the case-analysis and node-naming that they perform, can now be freely tailored to the needs of individual attributes; modifying the patterns will not interfere with the parsing of the program or with the ease of specifying other attributes. Clauses can be grouped in an ordering appropriate to a particular attribute definition, rather than that appropriate to the grammar. The vast majority of attributes that only deal with portions of the language can ignore other features of the language.

Pattern matching forms the primary method of specifying attribute equations in Sal, but the emphasis on the concept of an attribute equation specification language encourages experimentation in language extensions. For example, patterns are famed for providing excellent local context information, and notorious for the inability to perform trivial semantic checks. Thus, Sal has a simple mechanism for placing semantic preconditions on attribute equations. This section briefly introduces Sal's attribute equation specification language. Examples of many of the features can be found in Chapter 6.

4.3.1 `def-attribute` and pattern matching

The syntax of Sal is based on Lisp. The specification language is factored by particular attributes, instead of having a monolithic description. An attribute is defined with the `def-attribute` special form, specifying the name of the attribute and a list of clauses describing the equations defining the attribute:

```
(def-attribute attribute-name
   defining clauses)
```

The defining clauses describe the defining equations for the attribute named *attribute-name*. At the heart of each defining clause is a prototypical attribute equation with the prefix form `(:= (`*attribute-name node-expression*`)` *Lisp-form*`)`, defining the *attribute-name* attribute of the node *node-expression* to be equal to the value of *Lisp-form*. Around this attribute equation are wrapped restrictions on the bindings that can be taken on by the variables in *node-expression* and *Lisp-form*. These restrictions can involve pattern matching, semantic preconditions, and the enumeration of sets or lists.

The local context and case-analysis provided by productions in attribute grammars is provided by pattern matching in Sal. The `fa-matches` ("for all matches")wrapper has the form

```
(fa-matches (var pattern)
   defining-clauses).
```

The *defining-clauses* are valid for all bindings of the variable *var* and pattern variables in *pattern* such that *pattern* matches *var*. We can now define the calculator `val` attribute in Sal:

```
(def-attribute val
  (fa-matches (x ?.int)
    (:= (val x) (literal-val x)))
  (fa-matches (x [plus ?x1 ?x2])
    (:= (val x) (+ (val x1) (val x2))))
  (fa-matches (x [minus ?x1 ?x2])
    (:= (val x) (- (val x1) (val x2)))))
```

We are defining an attribute named `val`. At all nodes `x` that match the pattern `?.int`, that is, at all integers, the `val` attribute is defined to be equal to the `literal-val` of `x`; `literal-val` is a separately-defined function that obtains the integer value of a literal. At all nodes `x` matching `[plus ?x1 ?x2]`, the `val` attribute is defined to be the sum of the `val` attributes of the children.

4.3.2 The `default` clause

For many attributes, it is desirable to be able to state a default attribute equation to cover "the rest of the cases". Consider, for example, an attribute `left-sibling` intended to yield the left sibling of the child of an application:

```
(def-attribute left-sibling
  (fa-matches (some-application ({?} ?left-sib ?right-sib {?}))
    (:= (left-sibling right-sib) left-sib)))
```

Since the `{?}` notation signifies zero or more nodes, the DILS pattern

```
({?} ?left-sib ?right-sib {?})
```

will match a given application (with at least two children) once for each adjacent pair of children bound to `left-sib` and `right-sib`. This correctly sets up the definition of the `left-sibling` attribute for all nodes that actually have a left sibling. But what if we ask for the `left-sibling` of some node that is the first child of the application, or isn't a child of an application at all? The current definition would provide a run-time error; an alternative is to include a `default` clause, providing a default value (in this case `nil`) for nodes that have no other equation associated with them:

```
(def-attribute left-sibling
  (default nil)
  (fa-matches (some-application ({?} ?left-sib ?right-sib {?}))
    (:= (left-sibling right-sib) left-sib)))
```

4.3.3 Semantic preconditions

Semantic preconditions on the validity of a *defining-clause* may be enforced through the use of the `when` form:

`(when` *boolean-expression*
  defining-clauses`)`.

The *boolean-expression* is an arbitrary (side-effect free) expression; the *defining-clauses* are valid iff the *boolean-expression* is true. Semantic preconditions are useful when some condition cannot be expressed through the local context.

For example, the `lambda-depth` attribute, defined as the number of lambda expressions enclosing a given node, requires a test to make sure that a node has a parent (`tnode-parent` is a predefined function that returns the parent of a node, or `nil` if the node has no parent):

```
(def-attribute lambda-depth
  (default (lambda-depth (tnode-parent self)))
  (fa-matches (node ?any)
    (when (null (tnode-parent node))
      (:= (lambda-depth node) 0)))
  (fa-matches (lam (lm (?) ?body))
    (:= (lambda-depth body) (+ 1 (lambda-depth lam))))).
```

The `default` clause in this example references the node `self`; within a `default` clause, `self` refers to the node whose attribute is being defined.

In addition to the `when` form, several other Lisp-style selection forms (`unless`, `if`, and `cond`) are provided in a similar fashion.

4.3.4 Enumerators

In the traditional attribute-grammar specification, the attribute being defined in an attribute equation must belong to one of the nodes in the production. But it is frequently useful to specify an equation for a node that is *computed*, rather than represented in the production (or, in Dora, represented in the pattern); for example, one might want to specify an equation for an attribute of a node that is, itself, stored as an attribute value. Dora provides this functionality by allowing an arbitrary *node-expression*, rather than a simple node name, in the equation description form `(:= (`*attribute-name node-expression*`)` *Lisp-form*`)`.

This does not solve the problem of referring to arbitrarily computed *sets* of nodes. Sets of nodes described by the patterns they match can be handled with the `fa-matches` wrapper; arbitrarily computed sets are handled with the `fa-set-members` form:

```
(fa-set-members (var Lisp-form)
   defining-clause).
```

This form states that the *defining-clause* holds for all bindings of *var* to a member of the set returned by the *Lisp-form.* Suppose, for example, that `var-uses` returned the set of all uses of a particular variable. For each variable use, the lambda expression binding that variable can be defined with

```
(def-attribute lambda-binding-variable-in-use
  (default nil)
  (fa-matches (lam (lm (? ?v ?) ?body))
    (fa-set-members (use (var-uses v))
      (:= (lambda-binding-variable-in-use use)
          lam)))).
```

For all lambda expressions `lam` and all variables `v` of `lam`, for all uses `use` of `v`, the appropriate attribute equation is specified.

A similar form, `fa-list-members`, specifies that a given equation holds for all of the members of a list.

4.4 Handling recursive and multiple definitions

As in traditional attribute grammars, circular dependencies are not allowed in attributes defined with `def-attribute`. When circular dependencies are prohibited, there is a simple implementation method for solving the resulting set of equations. Furthermore, an unadorned `def-attribute` form that provides multiple or circular definitions may not have a well-defined solution to the resulting set of equations.

But circular dependencies are common in the straightforward specification of many data-flow problems, which are ubiquitous in code optimization. There have been some attempts at allowing circular definitions in attribute grammars [Far82, SEF89, WaJ88]. In Sal, this circularity is allowed through the use of a set of *defining inequalities.* The set of inequalities takes the role of the set of equations in the traditional framework; within the inequality framework, the restrictions against multiple and recursive definitions are lifted. Given such a description, Sal finds a least-fixed-point solution to the set of inequalities. The user must describe the lattice in which the solution is to be found by giving the bottom element, equality operator, and meet operator.

4.4.1 Recursive and set attributes

Attributes defined with `def-attribute` may not have recursive attribute definition functions. Instead, the `def-rec-attribute` form must be used:

```
(def-rec-attribute (name :test test-func :meet meet-func :bottom bottom-func)
  one or more defining-clauses)
```

The *test-func*, *meet-func*, and *bottom-func* are Lisp functions defining the equality predicate, meet operator, and a function returning the bottom value of the lattice to be used in finding a fixed-point solution to the attribute equations. There cannot be a `default` clause (the *bottom-func* effectively provides one); within the *defining-clauses* the `>=` form is used, rather than `:=`. `(>= (attrib node) val)` asserts that the value of `(attrib node)` is greater than or equal to the value of `val` within the lattice.

A common lattice used in recursive attribute definition has sets as values, with union as the meet operator. This lattice is common enough that a special form is provided for it, `def-set-attribute`:

```
(def-set-attribute name
  set-type
  one or more defining-clauses)
```

The *set-type* gives information about the kinds of elements that may be in the set; this information is used by Dora to give efficient implementations. The defining clauses again have different special forms in place of `:=` or `>=`: *includes* and *contains*. `(includes` x y`)` is simply a more descriptive name for `>=`, given that the meet operator is union; it states that the set x must include the set y. `(contains` x y`)` states that the set x contains the element y; it is equivalent to `(includes` x `(make-set` *set-type* y`))`.

As a simple example of `def-set-attribute`, consider the attribute `siblings`, the set of all siblings of the child of an application:

```
(def-set-attribute siblings
  *node-set*
  (fa-matches (x ({?} ?left-sib ?right-sib {?}))
    (contains (siblings left-sib) left-sib)
    (contains (siblings right-sib) right-sib)
    (includes (siblings left-sib) (siblings right-sib))
    (includes (siblings right-sib) (siblings left-sib))))
```

Once again, the `{?}` pattern matches zero or nodes, so the `contains` and `includes` clauses are valid for each adjacent pair of application children. Within an application, each `siblings` attribute contains its own node, plus all of the siblings of the node to its left and the node to its right.

4.4.2 Recursive description of a set of inequalities

We have been discussing a recursive set of inequalities, leading to a least-fixed-point solution. It is also possible to write a recursive *description* of the set of inequalities, by making the existence of some inequalities dependent on the values of some of the attributes. For example, the following clause occurs naturally when describing Shivers' control-flow analysis method for Scheme [Shi88]:

```
(fa-matches (app (?func {?args} ?arg {?}))
  (fa-set-members (lam (functional-values func))
    (fa-matches (use-of-var (^ !(nth-formal-parm lam (length args))))
      (includes (functional-values use-of-var)
                (functional-values arg))))).
```

Within this clause, the `functional-values` attribute for `func` is used to set up additional inequalities (`includes` relations, in this case) for other `functional-values` attributes.

This style of recursion is not allowed in `def-attribute`. Unrestricted usage in `def-rec-attribute` or `def-set-attribute` can lead to circular definitions without a unique least-fixed-point. Consider, for example, the following:

```
(def-set-attribute ill-defined
  *node-set*
  (fa-matches (node ?any)
    (unless (set-member node (ill-defined node))
      (contains (ill-defined node) node))))
```

In this example, the `contains` relation necessary for a node's inclusion in `ill-defined` is dependent on the node's non-inclusion; i.e., it's nonsense. Sal allows the user to write such circular definitions; it is up to the user of Sal to prove that a unique fixed point exists. A simple rule to follow is to ensure that raising the value of an attribute in the lattice does not remove inequalities, but only adds new ones. The former might remove the existence of a fixed-point, since raising an attribute value could remove an inequality that was necessary for the value to be raised in the first place.

In general, recognizing the places where using an attribute value might violate this restriction is difficult. But some simple cases can be recognized. In particular, in the lattice of sets with union as the meet operator, using an attribute value as the set in a `fa-set-members` form can only add inequalities as new members are added to the set. The motivating control-flow analysis example is of this form.

4.5 Derived nodes

DILS is a convenient intermediate representation for many purposes, but not for all. Sometimes a different view of the program is desired; for example, a control-flow graph is a traditional and useful view of a program for many optimization purposes.

To facilitate different views of this nature, it is possible to define *derived nodes*. Derived nodes are divided into classes; each node may have up to one derived node of the given class. The sole purpose of derived nodes is to have attributes associated with them through Sal. New classes may be defined by the user with the `def-derived-node-class` form:

`(def-derived-node-class` *class-name*`)`

The derived node of a given node n and class c is referenced by the form `(`c n`)`. Examples of the use of derived nodes will be given in Chapter 6 on UOPT.

4.6 Implementation

We have already noticed the importance of conceptually distinguishing the description of a set of defining equations from the engine to solve the described set. To ease experimentation with the language design, the implementation of Sal in Dora maintains this distinction within the code. Attribute descriptions are first evaluated in a relatively straightforward way to build a data structure representing the set of equations, and a separate engine then traverses this data structure to solve the equations.

Sal is built on top of Common Lisp, and it was desirable to use as much of Common Lisp's built in functionality as possible. For this reason, the implementation does not fully analyze Sal attribute definitions; forms such as `let` and `when` are directly implemented by the underlying Lisp. More importantly, it is possible to use Lisp's function and macro definition facilities with Sal. Doing so is very useful for language experimentation, since it greatly eases extension capabilities in a way that is impossible outside of the Lisp world, given our current language environment technology. But it raises the question of how Sal is able to determine the set of attribute equations and their dependencies on one another without static analysis. The answer is that Sal uses a fully general dynamic dependency technique. The technique could be replaced with a static technique in cases where execution speed was important.

4.6.1 Non-recursive attributes

When an attribute needs to be evaluated, the evaluator first executes the attribute definition as a regular Lisp body. The `fa-matches` form executes its subforms for all bindings of pattern variables providing a match; it is implemented using `tree-match-case`, and parallel uses of `fa-matches` are compiled into a single large `tree-match-case` to make use of the power of the automaton to match several patterns at once. When a defining clause `(:= (`*attribute node*`)` *value*`)` is evaluated, the *node* form is evaluated to yield a particular node, and a structure containing the attribute name and a closure over the *value* expression is associated with the node. Having computed this representation of the set of equations, work is handed off to the equation solver.

For non-circular attributes, the equation solver is trivial. When the attribute for a particular node is required for the first time, its associated attribute closure is evaluated and the returned value stored for future uses of the attribute. The node is marked before evaluating its attribute closure, so that recursive evaluations can be caught and marked as errors. Nodes without associated closures evaluate the `default` clause.

4.6.2 Recursive attributes

Recursive attribute evaluation in Sal is based on the general technique of "iterating until done", that is, initializing the attribute values to the bottom element of the lattice and then repeatedly interpreting inequalities of the form `(>= (attrib node) val)` to mean "set `(attrib node)` to `(meet (attrib node) val)`", where `meet` is the meet operator of the lattice. Assuming the lattice is finite (a required property for lattices used in `def-rec-attribute`), this iteration will eventually terminate with a fixed point solution to the set of inequalities. The reader unfamiliar with this traditional data flow analysis technique is referred to Kennedy's survey [Ken81].

The order in which inequalities are evaluated has a large effect on the speed of the iterative technique. Consider an inequality of the form `(>= (attrib n1) (attrib n2))`. If this inequality is evaluated just prior to evaluating an inequality modifying `(attrib n2)`, the first evaluation is essentially wasted. Ideally, evaluation of an inequality is delayed until all of the attribute values it depends on are fully evaluated; in this ideal situation, the inequality only needs to be evaluated once. Recursive dependencies make this ideal situation impossible, but approximations to the ideal situation are much more efficient than simply evaluating inequalities in a random order.

In traditional data-flow analysis problems, dependencies are closely related to the control-flow graph, and a good ordering can be obtained by evaluating the inequalities in an ordering based on a depth-first traversal of the control-flow graph. For Sal attributes, the relationship between the control-flow graph and the dependencies often does not exist. Sal therefore builds an explicit dependency graph showing the relationships among the inequalities, and chooses an evaluation order based on this graph.

Setting up the dependency graph

As in the non-recursive case, recursive attribute evaluation begins by executing the attribute definition as a regular Lisp body (after parallelizing `fa-matches` clauses using `tree-match-case`). Executing the form `(>= (attrib node) val)` constructs a *worklist-member*, consisting of the `node` and a closure over the `val` expression. This worklist-member is installed as the *current-member*, and the `val` expression is then evaluated and stored as the initial value of `(attrib node)`.

During this initial pass, when an expression of the form `(attrib node)` is evaluated, the current-member is added to a *dependency-set* for `node`, and `node` is added to a *local-worklist* for the current-member. Thereafter, whenever the value of `attrib` at `node` changes, all of the worklist-members in `node`'s dependency set are added to a global worklist, to be re-evaluated later.

Traversing the global worklist

After the initial pass, the global worklist will have worklist-members on it, placed there when the value of a node with dependents was changed. These worklist-members must be re-evaluated, as they depend on attribute values that have changed since the members' last evaluation.

The global worklist is organized as a FIFO queue. A particular worklist-member is in the list at most once; additions to the list of members already present is simply ignored. The list is traversed by taking the head of the global worklist and investigating its local-worklist (the set of nodes that the worklist-member depends on). If any of these nodes have worklist-members on the global worklist, these worklist-members are evaluated (in turn checking their local-worklists, and so on) and removed from the global worklist before the head element is evaluated. As in the initial pass, whenever a node's attribute value changes, its dependency-set is added to the global worklist.

The process of investigating the head of the global worklist and evaluating it, possibly after evaluating several other worklist members by searching through the dependency graph, is iterated until the global-worklist is empty. Since worklist-members are added to the global worklist only when an attribute value changes, and there are a finite number of possible attribute values, this process eventually terminates.

Performance of the algorithm

A precise analysis of this algorithm has not been performed. Such analysis is an interesting area for future work, and might reveal better choices in some areas. But as it stands, the algorithm is at least as good as the traditional iterative techniques based on depth-first orderings of the control-flow graph. The algorithm basically gives the same ordering as these traditional techniques for equivalent problems, since it performs a depth-first search of the dependency graph when evaluating worklist-members on the global worklist. It can perform better than the traditional iterative technique since worklist-members are only placed on the global-worklist when a value they depend on has changed; the traditional technique repeatedly evaluates all members, even when the changes are restricted to a small loop.

Chapter 5

Tess

Given a tree-oriented intermediate language such as DILS, a tree transformation language based on pattern-matching is a basic necessity for simple description of optimizations. Dora provides Tess, a tree transformation language that uses pattern-based tree rewrites extended with semantic preconditions in a style similar to Sal. In this chapter, we start by looking at typical tree transformation systems and hinting at major differences in Tess, and then give a fuller description of Tess with some simple examples.

5.1 Tree transformation systems

A *tree transformation specification* is a compact description of some relation between trees in an input language and an output language. Tree transformation systems provide a transformation specification language and an evaluation mechanism to find some output tree that is related to a given input tree.

5.1.1 Rewrite systems

The traditional notion of a *rewrite rule* is a pair of patterns α and β, written $\alpha \rightarrow \beta$, where α is called the input pattern and β is called the output pattern. The output pattern is not allowed to have pattern variables or wildcards, and any instantiation variables in β must appear as pattern variables in α.

A rewrite rule $r = \alpha \rightarrow \beta$ can be *applied* to a tree t if α matches t with exactly one set of variable bindings. The application of r to t, $r(t)$, is the instantiation of β using the bindings produced by the match.

A *rewrite system* is a collection of rewrite rules. A rewrite system specifies, for any given input tree t, a set of *subtree replacements* of the form $t_n \rightarrow t'_n$ where t_n is some subtree of t and t'_n is the result of applying some rule r in the system to t_n. The traditional notion of applying a rewrite system to a tree t is to repeatedly choose some tree replacement applicable to the current tree, perform the replacement, and then start over with the new tree. This notion of application yields systems equal in power to Turing machines; the tree can be used to simulate the tape, special operators introduced to represent the state and tape position, and each transition can be modeled with a simple rewrite rule.

5.1.2 The role of the rewrite rules in tree rewrite systems

We divided attribute systems into three main components: the specification of a set of attribute equations, the language used within the equations themselves, and the evaluation engine that solves the set of equations. Tree transformation systems can likewise be divided into three separate subsystems: the specification of a set of subtree replacements, the language used to describe the replacement tree, and the evaluation engine that performs the actual replacements.

In tree rewrite systems, the specification of the set of replacements and the replacement tree language are combined with the concept of the rewrite rule. The set of replacements is specified by the input patterns; each rule is applicable wherever its input pattern matches. The replacement tree language has two simple constructs, variable binding (provided by the input pattern) and tree construction through pattern instantiation (provided by the output pattern).

Tess separates these two functions of the rewrite rule. Instead of simply using pattern matching to specify the set of replacements, Tess uses the full power of Sal's specification language, including pattern matching, semantic preconditions, and enumerators. Tree replacements are specified using variable bindings and pattern instantiation, as in typical tree rewrite systems.

5.1.3 Performing tree replacements

Rewrite systems are equivalent in power to Turing machines. But this power is difficult to harness with the traditional application notion, where an arbitrary replacement is selected. The non-determinism introduced by this arbitrary selection can easily lead to non-terminating applications for many rewrite systems one would like to write. For example, adding a simple commutativity rule might lead the system to commute two arguments endlessly. Practical tree transformation systems need to provide a solution to this non-determinism.

One common solution is to exploit the non-determinism, by either finding classes of rewrite systems where all different application paths converge, or by specifying some particular set of goal trees and finding a particular application path that leads to a goal tree. Specifying a goal set is used by Pelegrí and Farnum in the context of code generation [Far88, Pel88]. This work leads to very fast systems, but limits the combinations of rewrite rules that may be used. For example, rewrite sequences that may pass information arbitrarily far down in the tree are not allowed.

Dora's emphasis on prototyping encourages ease of writing, and the limitations on rule use enforced by goal-oriented rewrite systems overly restricts rule usage. Therefore, Tess takes an alternative approach — non-determinism is limited by allowing the user to specify which replacements are to be chosen throughout the rewriting process. A simple and orthogonal declaration system allows the user to specify differing rule priorities, tree-walks, and re-application options. A similar approach is used in the OPTRAN transformation system [LMW88]. The resulting methods of choosing which rewrites are applied vary in power from simple *tilings* of the tree (where each node is rewritten at most once) to the turing-equivalent method of picking a rewrite, applying it, and repeating the process.

5.2 Defining tree transformations with Tess

5.2.1 Specifying subtree replacements

In Sal, a defining clause consists of several wrappers — `fa-matches`, semantic preconditions, and so on — restricting the applicability of a central prototypical attribute equation `(:= (`*attribute node*`)` *Lisp-form*`)`. Tess uses identical wrappers to restrict the applicability of a prototypical subtree replacement. The replacement form, with its associated wrappers, is called a *rewrite clause.*

A subtree replacement prototype in Tess has the form `(->` *node-form output-pattern*`)`. It states that the node obtained from evaluating *node-form* can be replaced by a node constructed by instantiating *output-pattern.* A rewrite clause is formed by wrapping forms restricting the value of the *node-form* around the replacement form. As a simple example, the following rewrite clause states that all nodes `a-simple-app` consisting of the application of a parameterless lambda can be replaced by the body of the lambda:

```
(fa-matches (a-simple-app ((lm () ?body)))
  (-> a-simple-app !body))
```

5.2.2 The `def-transformation` form and composing replacements

Tree transformations in Tess are defined with the syntax

```
(def-transformation name composition-rule
  1 or more rewrite clauses)
```

Our rewrite clause above can be used in a transformation:

```
(def-transformation empty-beta-reduce ()
  (fa-matches (app ((lm () ?body)))
    (-> app !body)))
```

This defines `empty-beta-reduce` as a function that, when applied to a tree, beta-reduces all applications of zero-variable lambdas within the tree.

Just as defining clauses in Sal are not imperative forms that assign values but rather descriptive forms of a set of attribute equations, the rewrite clauses of Tess are not imperative forms that perform rewrites, but descriptions of a set of possible replacements. The *composition-rule* determines how these individual replacements should be composed to form a complete transformation. There are a number of different factors that can combine to form a composition rule:

- Composition order
- Replacement priority at a single node
- Single-application vs. re-application
- Timing of replacement set computation

For most of these factors, the writer of a transformation can choose one of a number of different possibilities. These different choices are summarized in the `def-transformation` form by providing a list of keywords detailing each choice. The possible choices, and the keywords signifying them, are listed below.

Composition order

The rewrite clauses specify several different replacements at different places in the tree. Applying a replacement may change the applicability of other replacements, so the order in which replacements are applied can be significant. Replacements may be applied top-down or bottom-up, and left-right or right-left within child lists. At most one of the keywords `:left-right` or `:right-left`, and at most one of `:top-down` or `:bottom-up`, may be included in the composition rule to specify the order of composition. The default order is unspecified.

Replacement priorities

In many transformations, at most one replacement will be applicable at any given node. If there is more than one that could apply, some method must be given to choose between the alternatives. A static priority system is used within Tess; the first lexically apparent applicable replacement in the `def-transformation` form is chosen. In cases where more than one replacement at the same node is represented by the same lexical form, an unspecified replacement is chosen.

Single-application vs. re-application

After applying a rewrite, the tree-walk in search of the next replacement may be continued at the children or parent of the replacement tree, yielding a single-application strategy, or the walk may be restarted at the replacement node or further back, yielding a re-application strategy.

The `:dont-reapply` keyword specifies the single-application strategy. In this case, the walk is continued at the parent of the replacement node (for bottom-up walks) or the children (for top-down walks). When :dont-reapply is in effect, a given node can be replaced at most once, hence the name “single-application”.

The `:reapply-at-node` keyword specifies that the walk should be continued at the replacement node. In this case, a given node may be replaced by several successive rewrites before the walk is continued.

Timing of replacement set computation

The rewrite clauses of a `def-transformation` form provide a static definition of a set of replacements. Applying a replacement changes the tree and, therefore, the set of replacements defined through the rewrite clauses. Sometimes it is preferable to determine a fixed set of replacements applicable to the original tree, and continue to choose replacements from this set; at other times, it is preferable to recreate the set of replacements after each individual replacement is applied.

The usual semantics in tree-rewrite systems is to recompute the set of replacements each time a node is rewritten. This is specified in Dora with the `:recompute-replacements` keyword; after a replacement is applied, a new set of replacements is computed. The walk then continues (at a node specified by the other composition rule keywords) and will choose the next replacement from among those present in the new set.

In pure rewrite systems, "recomputing the set of replacements" can be done quite efficiently, since the set of replacements at a node is dependent only on the patterns that match there, and these patterns are discovered by the inherently incremental bottom-up automaton. In Tess, many other factors can contribute to defining a set of replacements; semantic checks may be performed, and the node at which a replacement is specified (that is, the value of the *node-expr* expression in the `(->` *node-expr pattern*`)` form) is not even necessarily related to any particular matched pattern. This makes recomputing the set of replacements an expensive proposition, and so Tess makes available the `:fixed-replacements` option.

Under the `:fixed-replacements` strategy, the set of possible replacements is evaluated *once* at the beginning of the tree walk. Applying a replacement r does not change the replacements available at nodes that are not directly affected by the replacement, that is, at nodes that are not newly created by the instantiation of the output pattern of r.

Although the `:fixed-replacements` strategy was originally motivated by efficiency considerations, it has turned out to be the desirable strategy from ease-of-use considerations more often than not. During the application of a Tess transformation, individual replacements may leave the tree in an inconsistent state; for example, a variable may be represented by a stack storage location in some portions of the DILS code and by a register location in other portions. The `:fixed-replacements` strategy allows the attribute programmer to concentrate on the state of the system before and after global transformations, rather than worrying about transient states.

5.2.3 Specification of composition rules

The keyword choices in a composition rule are orthogonal with the exception of `:fixed-replacements` and the `reapply` rules. The use of `:fixed-replacements` implies `:dont-reapply`, since meaningful reapplication requires checking for new replacements in the output tree.

Rather than explicitly listing the keywords for each transformation, a composition rule can be defined and then used in place of the list. The form
`(def-composition-rule` *rule-name keywords*`)`
where `rule-name` is a symbol and *keywords* a list of composition keywords allows *rule-name* to be used in place of *keywords* in a `def-transformation` form.

5.3 The implementation of Tess

Efficient implementations of tree-rewriting systems gain speed over naive implementations in two ways: first, by using fast pattern-matching techniques as discussed in Chapter 3, and secondly by analyzing the rewrite system and collapsing chains of possible rewrites into a single rule. The second method can sometimes be carried to the extreme of completely applying all useful rewrites in a single walk over the tree, with constant time spent at each node.

The implementation of Tess makes use of the first method, but not the second. The avoidance of sophisticated analysis of the rewrite system is based on three observations:

1. The pre-processing time involved in collapsing chains of rewrites can be extensive. In the context of Dora, where transformations are rewritten frequently and used for experimental compilations rather than frequent production runs, the pre-processing time is not worth the additional cost.

2. Pre-processing of the rewrite system is most useful when several rewrites may be applied at the same node. In the optimization transformations that form the bulk of work done within Dora, a transformation usually performs a single rewrite at a node, rather than the several rewrites common in code generation systems or functional program evaluators.

3. The semantic analysis involved in performing optimizations in Dora takes the vast majority of the processing time, so heavy investments in the transformation technology are not yet cost effective. Chow makes a similar observation in his dissertation [Cho83].

For these reasons, the implementation of Tess is relatively straightforward. Since the wrappers constraining Tess replacements are identical to those constraining Sal attribute equations, computing the set of replacements is done using the same code as that used to compute the set of equations for a non-recursive attribute; the body of the transformation is evaluated as a regular Lisp body, after compiling parallel uses of `fa-matches` into a large `tree-match-case` form to take advantage of the simultaneous matching capabilities of the pattern matcher. Evaluating the prototypical rewrite clause (`->` *node pattern*) associates the output pattern and the current bindings of any instantiation variables with the evaluated node.

After computing the initial set of replacements, a tree walk is performed in the order given by the composition rule. When a node n is found with an associated output pattern, the pattern is instantiated and the resulting node replaces the the node n in the tree. The replacement set is recomputed if requested by the `:recompute-replacements` keyword, and the tree walk is appropriately restarted at the new node or at its children/parent, depending on the composition rule.

Recomputing a set of replacements after a change is done in a manner similar to the initial computation, with the exception that any pattern-matching is done incrementally. The bottom-up table driven matcher yields a natural incremental algorithm; when a tree is rewritten, the state representing the patterns matching at its root can be recomputed starting at the instantiated variables in the output pattern, which have the same state they had prior to the rewrite. Any semantic checks, and any Sal attributes they depend on, must be reevaluated.

Chapter 6

UOPT

Dora has been used to prototype UOPT [Cho83], an existing optimizer, both as a proof-of-concept and as an aid to the Dora design process. UOPT is a global (intraprocedural) optimizer for UCODE, an intermediate language based on the Pascal PCODE with extensions for optimization. UOPT was originally written by Frederick Chow, and a descendent is currently in use on Digital's DECstation workstations [Dig89].

Using Dora to build some kind of prototype was clearly an important concern, especially given the observation that too many untested algorithms are already extant in optimization research. The decision to "prototype" an already extant compiler, rather than a new one, was made so that it would be possible to compare the results of the prototype with the results of the actual compiler, to save the trouble of designing a new optimizer, and to show that real compilers (rather than a toy design) could be prototyped with Dora. Once the decision to prototype an extant compiler was made, UOPT seemed to be a natural choice; it is currently in widespread use, it is described at a high level in Chow's dissertation, and it covers the vast majority of traditional optimizations in a relatively clean system. The UOPT prototype demonstrates that Dora is capable of handling traditional global optimizations for Pascal-style languages in the context of a lambda-calculus based IL with tree-oriented analysis and transformation languages.

6.1 The structure of UOPT

Before discussing the Dora prototype, we will discuss the overall structure of UOPT and the main features of its design. UOPT is intended to be a language- and machine-independent global optimizer; it is limited to rather traditional languages due to its intermediate language, but supports a wide variety of sequential target machines. The optimizations include a few simple algebraic transformations and a coloring register allocator, but are primarily of the "machine-independent" variety where code is moved from high-frequency to low-frequency areas of the program. The machine-independent optimizations are implemented by using data-flow analysis as much as possible, rather than by computing a few fixed attributes, such as def-use chains, and then using additional *ad hoc* analysis for each optimization. For example, using the ideas pioneered by Morel and Renvoise [MoR79], a code-motion optimization will use data-flow analysis to compute specific deletion and insertion points for individual expressions.

Dora has been used to implement both the algebraic optimizations and the machine-independent ones; this chapter will concentrate on the latter, as they represent the bigger challenge to the assertion that Scheme-based ILs and a tree-oriented system are appropriate for traditional optimization techniques.

6.1.1 UCODE

The intermediate language used in UOPT is UCODE, an extension to the widely-used PCODE Pascal-based intermediate language. UCODE is a machine language for a hypothetical stack machine, extended with registers and comments for annotation by the optimizer. The code is represented at an intermediate level. Array address calculations are exposed, but variable address calculation (including up-level references) and procedure calls are handled by single instructions. Control flow is handled with gotos and labels. Procedure entry and exit points are marked; procedure parameters and return values are passed via the expression stack.

Within UOPT, linear sequences of intermediate language instructions are converted into expression trees. Straight-line sequences of expression trees are collected into basic blocks, which are connected via gotos and labels into a traditional control-flow graph.

6.1.2 Basic blocks and local attributes

Some simple analysis and optimization within a basic block is performed as the control-flow graph is being created. Node numbering is used to find syntactically similar expressions; simple common-subexpression elimination within the basic block is performed using this information, and a global set of expressions used within the procedure is created to be used by the data-flow algorithms.

Each basic block is annotated with local data-flow attributes for the variables and expressions within the block. Typical attributes state whether a particular expression is available at the exit of the block, or whether a given variable is anticipated at the beginning of the block. This information is encoded in bitvectors, using the global sets created in the flow-graph construction phase for indexing.

6.1.3 Performing data-flow analysis

A given global optimization begins by solving a set of data-flow equations. The variables in the equations are sets of expressions or program variables having a certain property, such as being "available" (previously computed on all paths), at a given point in the control-flow graph. The equations specify the value of a variable for some basic block in terms of the local attributes of that block and the variables associated with its successor and predecessor basic blocks in the graph.

Since the control-flow graph generally has loops, the resulting set of equations is circular; for some of the problems, a greatest-fixed point solution is required, while others need a least-fixed point solution. In UOPT, a meet-over-all-paths solution is found using a traditional iterative technique [MuJ81]; the variables are initialized either to the empty set or to the complete set, and then the equations are repeatedly "evaluated" by performing the meet operation on the right-hand side and the current value of the variable on the left-hand side, obtaining a new value for the variable, until a fixed-point is found.

6.1.4 Transforming the code

Once the data-flow analysis is completed for a given optimization, the code is ready to be transformed. Transformations in UOPT are quite simple, generally consisting of deletion and/or insertion of expression trees computed through the data-flow analysis. This transformation is performed by *ad hoc* hand-coded routines.

6.2 The UOPT prototype

In the UOPT prototype, there have been two concerns pulling in opposite directions. The first is that the prototype be sufficiently similar to UOPT to make comparisons of the output code possible, to show that Dora can prototype traditional optimizers. The second desire is to make the attributes and transformations as widely applicable as possible, to show that these traditional optimizations can be applied successfully to more powerful languages. I have generally resolved these conflicting concerns by providing minimal extensions to the UOPT optimizations; the analysis performed is correct for all DILS programs, including those that use features not expressible in UOPT (such as continuation captures), but the analysis often makes worst-case assumptions for non-UCODE-style programs.

6.2.1 UCODE in DILS

The first decision to be made was how to model UCODE in DILS. One possibility would have been to define DILS operators corresponding to each UCODE operator directly, and to use a basic block oriented subset of DILS as explained in Section 2.2. For example, the statement `i := i + 1` is translated into the UCODE

```
LOD L S 1 8244 36         push contents of i on stack; most args are type information
LDC L 36 1                push constant 1 on stack
ADD L                     add top operands on stack, push result on stack
STR L S 1 8244 36         pop top stack item into i
```

With appropriate operator definitions, this could be expressed as the DILS expression

```
(e2 (lod l s 1 8244 36)
 (e2 (ldc l 36 1)
  (e2 (add l)
   (e2 (str l s 1 8244 36)))))
```

But doing so is overly restrictive. First of all, this format is not within the spirit of UOPT, which internally builds expression trees as its intermediate `representation` of the stack-oriented *language* UCODE. Furthermore, to help demonstrate the ability to write reusable optimizations, the work on UOPT should be applicable to instances of DILS code that do not conform to the restrictive syntax used in Section 2.2; non-conforming code may be created by other front-ends, or by intermediate transformations on the code. Thus, the

decision was made to retain the basic *level* of UCODE for individual operators, but use the more general methods within DILS for control and data flow. The increment statement is compiled into the DILS expression
`(assign i (gen+ (contents i) (make-long 1)))`.
Although this expression may, at first glance, seem to be at a "higher level" than the UCODE, it closely corresponds to UOPT's internal format; the expression forms a simple tree, with type information present (automatically added by the type inferencer, rather than explicitly placed in the code, but just as easily accessed). The main difference is that the UCODE has already fixed an address for the variable `i` while the DILS refers to `i` symbolically, as an operator defined in this program; but UOPT likewise creates an internal representation for "the variable at 8244" and uses this representation in its analysis, rather than relying on integer addresses.

By allowing all of the DILS constructs to be used in the IL, the attributes and transformations in the prototype can be useful for other languages using operators at roughly the same level, that is, with explicit arithmetic and array address calculations and implicit variable address calculations. A complete list of the operators defined for the UOPT prototype can be found in Appendix C.

6.2.2 Control flow analysis

Handling all of the control-flow operators of DILS in the prototype makes building the control-flow graph a non-trivial problem. In pure UCODE, targets of gotos are always explicitly listed in the code; in DILS, gotos are represented by procedure calls, where the called procedure is the result of an expression evaluation and so may be bound to a variable or even fetched from the store. Determining the possible targets of a procedure call has always been a difficulty for Scheme-like intermediate languages.

Control-flow analysis for CPS code

Shivers has presented a solution for Scheme code that uses continuation-passing-style (CPS) control flow [Shi88]. In CPS code, the arguments of an application are severely limited: they may only be constants, lambda-expressions, or variable references. Nested expressions are computed by passing an extra argument, an explicit continuation, to all operators; an operator computes its usual value and then passes this value as an argument to its continuation. The implicit continuation in the usual nested expression tree is replaced with the explicit continuation in CPS. For example, the expression `(plus x (plus y z))` would be represented in CPS code as `(plus y z (lambda (y-plus-z) (plus x y-plus-z` k`)))` where k is the continuation of the entire expression, that is, what we wish to do with the sum.

Shivers solves the control-flow-analysis problem by defining the set Defs(e), the set of all lambda expressions or operators to which the expression e might evaluate. Strictly speaking, an expression cannot evaluate to a lambda expression, but instead can evaluate to a closure. Shivers treats all of the closures generated by a given lambda expression as the same object, since a given lambda expression might evaluate to an unbounded number of closures (with different sets of bindings). This treatment provides the traditional interpretation of the control-flow graph, where basic blocks are differentiated by different

```
(def-set-attribute cps-defs *node-set*
  (fa-matches (lam ?.lam)
    (contains (cps-defs lam) lam))
  (fa-matches (op ?.op)
    (contains (cps-defs op) op))
  (fa-matches (variable-use (^ ?v))
    (includes (cps-defs variable-use) (cps-defs v)))
  (fa-matches (app (?func {?args}))
    (fa-set-members (func-val (cps-defs func))
      (when (lambda-p func-val)
        (fa-list-members (var/arg (pairlis args (lambda-formals func-val)))
            ;;i.e., for each formal parameter and corresponding actual argument
          (includes (cps-defs (car var/arg)) (cps-defs (cdr var/arg)))))))
  (fa-matches (app (Y (lambda ({?vars} ?k)
                              (?use-k {?funcs.lam}))
                      ?cont))
    (includes (cps-defs use-k)
              (cps-defs cont))
    (fa-list-members (var/func (pairlis vars funcs))
      (contains (cps-defs (car var/func)) (cdr var/func)))))
```

Figure 6.1: `cps-defs` for CPS Scheme

underlying code, but not by the different variable bindings that might exist at different points in time.

Shivers provides a recursive set of equations for Defs(e) in terms of the syntactic context of e and in terms of the Defs of the subexpressions of e; extracting the control-flow graph from a least-fixed-point solution to the set of equations is then trivial. Shivers' equations are summarized in the Sal attribute `cps-defs` defined in Figure 6.1.

The first three clauses of `cps-defs` spell out simple cases: lambda expressions and operators evaluate to themselves, while variable references might evaluate to any of the `cps-defs` of the variable referred to.

The next clause handles normal applications. Suppose `app` is bound to an application `(^func ^x ^y)`. Then, for each lambda expression `fv` that `^func` might evaluate to, the first variable of `fv` should include the `cps-defs` of `^x` and the second variable should include the `cps-defs` of `^y`. This clause contains a recursive use of `cps-defs` to add additional relations to the set of attribute inequalities; `(cps-defs func)` may increase in size as the evaluation proceeds if `func` is bound to a variable reference.

The final clause handles the `Y` operator, which Shivers uses for recursion in his CPS Scheme. The details of the operator are not important for our purposes; the clause simply adds the lambdas in `funcs` to the `cps-defs` for the variables in `vars`.

Control-flow analysis for DILS

Shivers' solution is quite clean; unfortunately, it assumes that we are enforcing the CPS convention. In order to use full DILS, we must add extensions to handle nested applications, more powerful operators, labels, and continuation captures/applications. These extensions are incorporated in the attribute `func-vals`. Like `cps-defs`, the `func-vals` attribute is the set of all expressions or operators that an expression might evaluate to; the definition of `func-vals` handles all of DILS, while `cps-defs` is restricted to CPS code. The complete attribute definition appears in Figure 6.2. We will examine the clauses of this definition in the following sections; each section begins with a copy of the clauses to be discussed, so that the reader will not have to continually flip to Figure 6.2.

Labels, operators, variables, and lambdas

```
(fa-matches (lab ?.lab)
  (let ((lam (label-lambda (label-instance-label lab))))
    (if lam
        (contains (func-vals lab) lam)
        (contains (func-vals lab) lab))))
(fa-matches (o ?.op)
  (contains (func-vals o) o))
(fa-matches (variable-use (^ ?v))
  (includes (func-vals variable-use) (func-vals v)))
(fa-matches (variable-ref (& ?v))
  (contains (func-vals v) *the-unknown-func*))
(fa-matches (lam (lm ({?vars}) ?))
  (contains (func-vals lam) lam)
  (when (escaped-func-val? lam)
    (contains (func-vals v) *the-unknown-func*)))
```

The clauses for the primitive function objects are straightforward; lambdas and operators can evaluate to themselves, and again variable references can evaluate to whatever the variable might be bound to. Some labels (those defined locally) have associated lambda expressions, which can be used directly. Very little is known about labels defined elsewhere; in these cases, the node `*the-unknown-func*` is placed in the `func-vals` set. `*the-unknown-func*` is included wherever an unknown function might result, and worst-case assumptions are made when it is seen in a set.

The last two clauses take care of variables that may be bound in unknown portions of the program. If a variable's address is used, then it may have something placed in it from the store; we simply add `*the-unknown-func*` at this point (some additional analysis could catch some simple cases). Some lambdas "escape" to unknown portions of the program, that is, they may be passed as arguments to external functions, or they may be placed in the store and fetched from unknown locations. Variables belonging to these lambdas may be bound to arbitrary values, and so we must put `*the-unknown-func*` in their `func-vals` set; the attribute `escaped-func-val?` (defined elsewhere) is true for functional values that escape, and false otherwise.

Applications

```
(fa-matches (app (?f {?args}))
  (fa-set-members (fv (func-vals f))
    (cond ((lambda-p fv)
           (includes (func-vals app) (func-vals (lambda-body fv)))
           (fa-list-members (var/arg (pairlis args (lambda-formals fv)))
             (includes (cps-defs (car var/arg)) (cps-defs (cdr var/arg)))))
          ((label-instance-p fv)
           (when (label-external? (label-instance-label fv))
             (contains (func-vals app) *the-unknown-func*)))
          ((eq fv *the-unknown-func*)
           (contains (func-vals app) *the-unknown-func*))
          ((op-instance-p fv)
           (cond ((op-prop (op-instance-op fv) :returns)
                  (fa-list-members (arg-pos (op-prop (op-instance-op fv)
                                                     :returns))
                    (includes (func-vals app)
                              (func-vals (nth arg-pos args)))))
                 ((op-prop (op-instance-op fv) :funcalls)
                  (fa-list-members (arg-pos (op-prop (op-instance-op fv)
                                                     :funcalls))
                    (fa-set-members (fv2 (func-vals (nth arg-pos args)))
                      (if (lambda-p fv2)
                          (includes (func-vals app)
                                    (func-vals (lambda-body fv2)))
                          (contains (func-vals app)
                                    *the-unknown-func*)))))
                 (t
                  (contains (func-vals app) *the-unknown-func*)))))))
```

The nesting of applications actually adds very little to the complexity of the control-flow analysis problem; the complexity of this clause, compared to the corresponding clause in `cps-defs`, is due to the addition of labels, `*the-unknown-func*`, and a general handling of primitive operators. These features (present in CPS code) were ignored in Figure 6.1.

When we look at an application, we must investigate each of the possible functional values of its function expression. For lambdas, we again add the `func-vals` of the actual arguments to the `func-vals` of the formal arguments. For labels and `*the-unknown-func*`, we must add `*the-unknown-func*` to the `func-vals` of the application (since we don't know what these applications might return).

Application of operators are the most difficult to handle, since we need some generic way of dealing with operators such as `if` that might call their operands as functions. I have chosen to associate two properties with each operator, `:returns` and `:funcalls`, to help in this problem. An operator with a non-null `:returns` property directly returns one of its operands, indicated by the property value; for example, the `prog2` operator has a `:returns` property with the value 1 (argument indices are zero-based). Since operators like prog2 are useful for inserting code at arbitrary points in a DILS program, it is important for the analyzer to take note of the :returns property to trace functional values.

The :funcalls property, when non-null, states that the operator calls one of its arguments and returns the returned value. The property value is a list of the arguments that might be called; thus, the :funcalls property for the if operator is set to the list (1 2), indicating that it will return the result of calling either the *then* thunk or the *else* thunk.

Continuation captures and applications

```
(fa-matches (capture (c-cap ?func))
  (fa-set-members (fv (func-vals func))
    (if (lambda-p fv)
        (let ((v (first (lambda-formals fv))))
          (contains (func-vals v) capture)
          (includes (func-vals capture) (func-vals (lambda-body fv))))
        (contains (func-vals capture) *the-unknown-func*)))
  (when (escaped-func-val? capture)
    (contains (func-vals capture) *the-unknown-func*)))
(fa-matches (cont-application (c-app ?cont-exp ?val))
  (fa-list-members (cont (func-vals cont-exp))
    (when (continuation-capture-p cont)
      (contains (func-vals cont) (func-vals val)))))
```

In a continuation capture, we simply need to add the continuation as a functional value of the parameter of func. The rest of the func-vals definitions correctly trace continuation captures when they are passed as parameters. We do not have to deal with continuation captures as possible elements of func-vals in the previous clause for applications since the type system ensures that continuations can only occur within continuation-applications.

The functional values for a continuation-capture could come either from the body of the capture itself, tested by examining the func-vals of the func, or from a continuation-application, tested in the second clause. As with lambda expressions, a test must be made for continuations that escape to unknown portions of the program.

Recursive label definitions

```
(fa-matches (lbs (labels ({?funcs}) ?body))
  (includes (func-vals lbs) (func-vals body)))
```

The labels expression returns whatever is returned by the body.

6.2.3 Creating the control-flow graph

With the func-vals attribute, it is relatively straightforward to determine how control can pass through the program. The control flow graph is built on top of the DILS representation by using derived nodes. With each DILS node, we associate at least two derived nodes, one representing the point just prior to evaluation of the node, and the other representing the point just after evaluation of the node. These derived nodes are named cp-before and cp-after, where cp is short for code-point. For applications, it is useful to define a third code point, the app-cp, that may represent the point of execution just before

```
(def-set-attribute func-vals *node-set*
  (fa-matches (lab ?.lab)
    (let ((lam (label-lambda (label-instance-label lab))))
      (if lam
          (contains (func-vals lab) lam)
          (contains (func-vals lab) lab))))
  (fa-matches (o ?.op)
    (contains (func-vals o) o))
  (fa-matches (variable-use (^ ?v))
    (includes (func-vals variable-use) (func-vals v)))
  (fa-matches (lam (lm ({?vars}) ?))
    (contains (func-vals lam) lam)
    (when (escaped-func-val? lam)
      (fa-list-members (v vars)
        (contains (func-vals v) *the-unknown-func*))))
  (fa-matches (app (?f {?args}))
    (fa-set-members (fv (func-vals f))
      (cond ((lambda-p fv)
             (includes (func-vals app) (func-vals (lambda-body fv)))
             (fa-list-members (var/arg (pairlis args (lambda-formals fv)))
               (includes (cps-defs (car var/arg)) (cps-defs (cdr var/arg)))))
            ((op-instance-p fv)
             (cond ((op-prop (op-instance-op fv) :returns)
                    (fa-list-members (arg-pos (op-prop (op-instance-op fv)
                                                       :returns))
                      (includes (func-vals app)
                                (func-vals (nth arg-pos args)))))
                   ((op-prop (op-instance-op fv) :funcalls)
                    (fa-list-members (arg-pos (op-prop (op-instance-op fv)
                                                       :funcalls))
                      (fa-set-members (fv2 (func-vals (nth arg-pos args)))
                        (if (lambda-p fv2)
                            (includes (func-vals app)
                                      (func-vals (lambda-body fv2)))
                            (contains (func-vals app)
                                      *the-unknown-func*)))))
                   (t
                    (contains (func-vals app) *the-unknown-func*))))
            ((label-instance-p fv)
             (when (label-external? (label-instance-label fv))
               (contains (func-vals app) *the-unknown-func*)))
            (t
             (contains (func-vals app) *the-unknown-func*)))))
  (fa-matches (capture (c-cap ?func))
    (fa-set-members (fv (func-vals func))
      (if (lambda-p fv)
          (let ((v (first (lambda-formals fv))))
            (contains (func-vals v) capture)
            (includes (func-vals capture) (func-vals (lambda-body fv))))
          (contains (func-vals capture) *the-unknown-func*)))
    (when (escaped-func-val? capture)
      (contains (func-vals capture) *the-unknown-func*)))
  (fa-matches (cont-application (c-app ?cont-exp ?val))
    (fa-list-members (cont (func-vals cont-exp))
      (when (continuation-capture-p cont)
        (contains (func-vals cont) (func-vals val)))))
  (fa-matches (lbs (labels ({?funcs}) ?body))
    (includes (func-vals lbs) (func-vals body))))
```

Figure 6.2: `func-vals` for full DILS

or just after application of the function the already evaluated arguments. An example of the use of `app-cp` will be given in Section 6.2.5.

Links in the control-flow graph between one code point and another are computed by two attributes of code points, `cp-succs` and `cp-preds`. `(cp-succs x)` is the set of possible successor code points of the code point `x`, and `(cp-preds x)` is the set of possible predecessors. The definition of `cp-succs` is relatively straightforward, except for dealing with the different possible types of `func-vals` in an application. A sample clause from the definition connects adjacent children of an application:

```
(fa-matches (app (? ?x ?y ?))
  (contains (cp-succs (cp-after x))
            (cp-before y)))
```

Iterating over all of the code points in a tree is done by using a pattern to iterate over all nodes, and an enumerator to iterate over a list of each node's code points. For example, `cp-preds` is defined simply as:

```
(def-set-attribute cp-preds *node-set*
  (fa-matches (node ?)
    (fa-list-members (cp (all-cps node))
      (fa-set-members (succ (cp-succs cp))
        (contains (cp-preds succ) cp)))))
```

where `all-cps` is a trivially defined attribute listing the code points for an expression.

6.2.4 Basic blocks and local attributes

Basic blocks serve two purposes in traditional optimization: they delineate straight-line code areas where simple and efficient analysis can be performed, and they make a convenient repository for summarized data-flow information to be propagated throughout the control-flow graph. In an optimizer using global data-flow analysis, both of these considerations are profitable only so far as speed of the optimizer is concerned. Having a simple local analysis, in addition to the more complicated global analysis, is only better if it is faster. Likewise, the summary of local data-flow information makes the global analysis faster, not better.

Since the purpose of the UOPT prototype is to build an optimizer that provides the same functionality as UOPT in the simplest way, basic blocks are not constructed. Alternatively, one can view each code point as its own basic block. The standard global data-flow equations handle straight-line sequences of basic blocks correctly, so the prototype gains simplicity by having a single kind of analysis instead of separate "global" and "local straight-line" portions. Some local analysis is still needed, of course; for example, determining which variables are modified by a given application (modulo alias effects) is a local analysis problem, not a global data flow problem. But by uniting the global data-flow and local straight-line analyses, the remaining local analysis can be performed within the global data-flow description when appropriate. An example of this kind of analysis will be seen in the following section.

The efficiency lost by handling straight-line code with the general global solution is noticeable in the prototype; long straight-line sections of code that do very little actual computation (for example, safe but unnecessary code introduced by earlier transformations) have a significant impact on the speed of analysis. One possible solution to this problem in future prototypes would be to construct basic blocks using derived nodes, in a manner similar to constructing the control-flow graph, and collect summary information. A better solution would involve an enhancement to Sal to recognize copied attributes and share the resulting data structures, rather than copying them from node to node.

6.2.5 Performing data-flow analysis

Data-flow analysis is performed using Sal. Data-flow analysis problems are generally described in the literature by describing a system of equations over basic block nodes, and it is a straightforward process to convert these descriptions into Sal attributes on code-points. For example, Chow defines the *partial availability* attribute for expressions as follows [Cho83]:

> An expression is *partially available* at a given point if at least one path leading to the point contains the computation, and the computation placed anywhere along the path always delivers the same result.

He gives the following set of equations to compute the sets of partially available expressions:

$$\begin{aligned} P_I(x) &= \sum_{y \in \mathrm{Pred}(x)} P_O(y) \\ P_O(x) &= G(x) \cup (\neg K(x) \cap P_I(x)) \end{aligned}$$

where

- $P_I(x)$ is the set of expressions partially available at the entry to basic block x,
- $P_O(x)$ is the set of expressions partially available at the exit of x,
- $G(x)$ is the set of expressions computed with x that are available at its exit, and
- $K(x)$ is the set of expressions *killed* by x, that is, those expressions that depend on variables modified within x.

An "expression" here is a mathematical tree, not a portion of IL code. The DILS node `(plus (plus ^a ^b) (plus ^a ^b))` has two arguments, internally represented by two different structures, but each argument is mathematically speaking the same tree. In the UOPT prototype, these mathematical trees are called *treeids*, and the function `tnode-treeid` returns the treeid for a given node. This distinction, often ignored in the literature, accounts for assorted conversion functions between treeids and nodes in the following examples.

Since UOPT uses basic blocks, Chow must distinguish between the the entry point of a block, where information from previous blocks converges, and the exit point, where the information at the entry is modified by the activity within the block. In the Dora prototype, these two different influences are modeled by treating normal code points (those associated with the timing before and after an expression is executed) with the `app-cp`, the code point associated with the actual application of a function to its arguments. At normal code points, information is simply propagated by the meet operator; generation of new information, or killing of old information, is done at the application code points. This is done as follows for the `pavailable` attribute, which computes the set of partially available expressions at a code point:

```
(def-set-attribute pavailable
  *tree-set*
  (fa-matches (x ?)
   (fa-list-members (cp (b/a-cps x))
    (fa-set-members (pred (cp-preds cp))
     (includes (pavailable cp)
               (pavailable pred)))))
  (fa-matches (app ?.app)
   (let ((cp (app-cp app)))
    (fa-set-members (pred (cp-preds cp))
     (includes (pavailable cp)
               (strip-killed-exprs app (pavailable pred))))
    (when (and (not (and (some-var-escapes? (cell-support app))
                         (set-member *the-unknown-cell* (cells-modified app))))
               (not (set-intersect-p (cells-modified app)
                                     (cell-support app))))
     (contains (pavailable cp) (tnode-treeid app))))))
```

The first clause looks at all of the normal (before/after) code points of all expressions, and states that the `pavailable` set contains everything in the `pavailable` sets of the predecessor code points.

The second clause deals with applications, where new expressions may become available and old ones may be killed. The subclause

```
(fa-set-members (pred (cp-preds cp))
 (includes (pavailable cp)
           (strip-killed-exprs app (pavailable pred))))
```

propagates the `pavailable` sets of the predecessors through the application, with the `strip-killed-exprs` function removing expressions that are killed by the application. This subclause plays the part of the $\neg K(x) \cap P_I(x)$ expression in Chow's presentation.

The subclause

```
(when (and (not (and (some-var-escapes? (cell-support app))
                     (set-member *the-unknown-cell* (cells-modified app))))
           (not (set-intersect-p (cells-modified app)
                                 (cell-support app))))
 (contains (pavailable cp) (tnode-treeid app)))
```

adds the application to the `pavailable` set when appropriate. The application should not be added if it contains a variable that *escapes*, that is, a variable that might be aliased with `*the-unknown-cell*`, when `*the-unknown-cell*` is modified; nor should it be added if it modifies one of the cells that it uses. Otherwise, the application can be safely added to the `pavailable` set.

For this attribute, the `app-cp` represents the code point just after the application of the function, and so represents the same location as the `cp-after` code point; but this separation allows us to use the simplified clause for all before/after code points, rather than inserting an explicit check into that clause for the code points after an application. For attributes that pass information backwards through the control-flow graph, it is usually appropriate to have the `app-cp` represent the point just prior to the application, but after the evaluation of sub-expressions; this again allows all before/after code points to be handled separately.

Normal Lisp code can be present in the attribute definition, as long as it is functional. This is quite useful when dealing with the local analysis that must be performed (the analog of creating the sets $G(x)$ and $K(x)$ in the traditional formalism). The usual Lisp abstraction mechanisms can be used when appropriate (for example, encapsulating the relatively complicated checking for killed expressions in the `strip-killed-exprs` function, which is used in several other attributes), while simple analysis can be done inline without destroying the readability of the attribute definition. Throughout the construction of the prototype, I have found this ability to mix Sal, Tess, and Lisp code to be an advantage; although experiments were done in creating higher-level macros that accepted a notation closer to the formalism used in the P_I equations, the rigidity of these macros made them much less useful than the underlying `def-attribute` facility.

6.2.6 Transforming the code

Once data-flow analysis has been performed, it must be used to modify the code. The transformations applied are generally quite simple, and have not stressed the capabilities of Tess; UOPT is a data-flow analysis centered optimizer. For completeness, a sample transformation is show in Figure 6.3 that performs global common-subexpression-elimination. The transformation has three tasks: it must store expressions that are to be reused in temporaries, it must replace expressions that have already been computed from their temporaries, and it must allocate the temporaries at an appropriate location.

Each tree has a temporary variable associated with it at a given point via the `temp-map`; these maps vary from region to region, and are stored at each lambda. If an application is `available` before a code point, it has been computed along all paths leading to that point and so can be replaced by a fetch from its temporary. If an application is `panticipated`, then it is recomputed along at least one path leading from this computation, and so must be stored in the associated temporary. Finally, at each lambda, we determine which temporaries are to be allocated at that lambda (using the `cse-trees-to-alloc` attribute) and do the allocation using the `uninit-storage` operator, which invokes a lambda by passing it undefined values (at a lower level, this will be transformed into code that simply modifies the stack pointer).

```
(def-transformation do-cse-stores-and-fetches ()
  (fa-matches (app ?.app)
    (let ((tree (tnode-treeid app)))
      (when tree
        (cond ((set-member tree (available (cp-before app)))
               ;;replace with proper fetch
               (let ((temp (lookup tree (temp-map (parent-lambda app)))))
                 (-> app (^ !temp))))
              ((set-member tree (panticipated (cp-after app)))
               ;;store in proper var
               (let ((temp (lookup tree (temp-map (parent-lambda app)))))
                 (-> app (e-assign (& !temp) !app)))))))))
  (fa-matches (lam (lm ({?vars}) ?body))
    (let ((temps '()))
      (do-set (tree (cse-trees-to-alloc lam))
        (push (lookup tree (temp-map lam))
              temps))
      (when temps
        (-> lam (lm ({!vars})
                    (uninit-storage (lm ({!temps})
                                        !body))))))))
```

Figure 6.3: A sample transformation from the prototype

6.3 Comparing the prototype and UOPT

In order to show that the prototype does, in fact, perform optimizations similar to UOPT, it is necessary to run the prototype on some real programs and compare the results. This involves building a compiler that produces suitable DILS code to be optimized and generates appropriate machine code for a target machine, in this case the DECstation.

While constructing the prototype, hand-written DILS examples were used for testing purposes. I had originally hoped to use a local Modula-2 front-end to produce high-level DILS, lower this DILS to a UCODE-equivalent level, apply the optimizations, and then produce machine code. The resulting system would help demonstrate the use of Dora in building a complete back-end, including important vertical transformations in addition to the horizontal transformations of the UOPT prototype. Consequently my initial design contained specifications of DILS operators for Modula-2.

Unfortunately, problems with the local front-end consumed too much time to make the construction of this compiler possible. So, at the last moment, a different decision was made: to take UCODE generated from the DECstation C compiler, convert it to DILS, optimize this DILS with the prototype, convert back to UCODE, and use the DECstation C compiler to generate the machine code from the optimized UCODE. This decision has had two important benefits. First of all, I gained the experience of converting the entire optimization suite to a completely different set of operators, and will discuss this experience in Section 6.3.2. Secondly, the benchmark timings have greater validity: by using the prototype as a substitute for the DECstation UOPT descendent, the only variable in the test timings is the optimizer itself. Direct comparisons with the numbers in Chow's dissertation is not feasible, since neither the compilers nor the machines he used are currently available.

6.3.1 Comparison of the overall architecture

The prototype has the same overall phase structure as Chow's optimizer, although there are some minor differences. UOPT's phases, and differences in the prototype, are:

1. Performance of local optimization. In UOPT, a handful of local optimizations are performed at this time. The prototype performs those optimizations that are not subsumed by later optimizations, but leaves out some (for example, local common subexpression elimination) that are simply local versions of the global optimizations to come.

2. Collection and setting up of data flow information. UOPT identifies syntactically-identical subexpressions at this point, sets up global tables for use in its bitvector set implementation, and collects summary local data flow information. This phase is not included in the prototype; syntactically-identical subexpressions are discovered when they are created, Dora's set implementation does not require global tables, and our optimizations do not use local summary information.

3. Processing of the program control flow graph. Although Dora must perform more work to find the control flow graph for the more general DILS structures, the result is the same.

4. Global copy propagation.

5. Redundant assignment elimination.

6. Partial redundancy elimination for expressions by backward code motion, including strength reduction. In UOPT, this phase marks expressions to be inserted and deleted in various blocks, but does not actually do the transformation; the insertions and deletions may be “undone” by further phases. This destroys the independence of phases desirable in Dora, so the prototype performs the actual transformation. This independence cannot be completely eliminated, as the next phase (linear test replacement) makes use of information regarding the strength reductions done by this phase. In the prototype, this information is passed on by inserting operators into the tree. Semantically, these operators are identity operators, so they do not impact the operation of other optimizations.

7. Linear test replacement. In UOPT, this optimization makes use of information still present from the partial redundancy elimination phase; in the prototype, the operators inserted by the previous phase pass along the information. If these operators are not present (for example, if the previous phase was not performed), the transformation passes the code through unchanged.

8. Induction variable elimination. In UOPT, I believe this optimization finally makes the transformations that have been decided on in the previous two phases, which mark expressions to be inserted and deleted in various blocks. In the prototype, the internal representation is transformed by each phase, and so the tree is already up to date; this reduces this phase to a second application of the redundant assignment elimination optimization.

9. Partial redundancy elimination for stores by forward code motion.

10. Global register allocation. This phase marks the major departure from UOPT; the register allocator in the prototype is not as powerful as the UOPT allocator. Live range splitting and taking the depth of loop-nesting into account are not done by the prototype. These lacks are primarily due to time constraints; the other optimizations have several concepts in common, and so techniques (and often code) used in implementing one can be reused in another, while time spent on the register allocator did not reap dividends elsewhere. The register allocator as it stands is sufficient for the simple benchmarks given in Chow’s thesis, which do not stress the large number of registers available on the MIPS chip.

11. Emission of optimized code. UOPT performs some unspecified local transformations to some op-codes, presumably of a peep-hole nature. The prototype does not perform any transformations at this point. Due to the RISC-style architecture of the MIPS chip, these optimizations do not seem likely to be of great importance.

6.3.2 Converting to UCODE

The prototype was written and tested using an IL at the approximate level of UCODE, but with a decidedly different style; control-flow was represented using DILS `labels` expressions and function calls, type information was represented using the type-inferencer, and most importantly source-level variables were represented using lambda variables. In UCODE, control-flow is represented using jumps, type information is represented by explicit operands in the code, and load and store operations use explicit addresses in accessing memory.

Handling the different set of operators was relatively straightforward; simple descriptions were added to define the number and type of expected operands. Reading the UCODE and converting it to the DILS internal representation was easily done since the DECstation compiler can produce and consume UCODE in ASCII format. The control-flow style of UCODE was taken care of by using DILS in a basic-block style, as described in Section 2.2; since this is simply a coding convention, rather than a different set of DILS semantics, the control-flow analysis written for full DILS is equally applicable to this restricted usage.

Having UCODE in DILS, one major change and a few minor changes had to be made to the prototype. The major change involved the storage model. The prototype only traced changes to the cells bound to lambda variables, and lumped all other stores and fetches into a single category, handling them with conservative assumptions. In UCODE, memory is broken up into four different areas: static storage, parameter storage, procedure local storage, and registers. Variable references are converted into addresses within these different storage areas. The prototype had to be modified to take into account this more complicated storage model; doing so involved creating a new Lisp-type, `ucode-cell`, and modifying the Sal attributes that traced the creation, use, and deletion of variable cells to handle the new type of cell. The prototype is sufficiently modularized that this could be done relatively easily, even though virtually all of the high-level UOPT optimizations depend intimately on tracing how information flows through the store.

Writing the code to read and write UCODE and a first cut at the new storage model took one day. Programs were being successfully optimized and running within two days. Over the following week, the optimizations were debugged running over Chow's benchmark suite, linear test replacement was added to the prototype, thereby completing the major optimizations in UOPT, and enhancements were added to the register allocator. The ability to convert an optimizer the size of UOPT to a new IL and debug it over real programs in the span of a week attests to the usefulness of Dora in experimenting with optimizers.

6.3.3 Benchmark results

Chow reported the results of optimizing thirteen benchmark programs with UOPT. I have successfully compiled nine of these programs; although originally written in Pascal, they do not make use of procedure nesting and have been transliterated to C. The nine programs compiled were:

Perm — A program that permutes the values of an array using recursion.

Tower — A recursive solution to the Tower of Hanoi problem.

Queen — A recursive solution to the Eight Queens problem.

Intmm — Multiplication of two 40 by 40 integer matrices.

Puzzle — A brute-force solution to a bin-packing problem.

Quick — A quick-sort routine.

Bubble — A bubble-sort routine.

Tree — Performance and checking of inserting nodes into an ordered tree.

Sieve — Sieve of Erastosthenes computation of primes.

The four programs that were not compiled were written in Fortran and/or made use of floating-point arithmetic. I could not determine how to make use of the floating-point registers in the DECstation compiler (Chow's documentation of UCODE only contains information on the machine-independent features), so optimization of these programs was not possible.

One of the experiments Chow performed was to use PMERGE, a procedure merging routine written at Stanford, to inline-expand the programs before optimization. Since several of the benchmarks make heavy use of procedures (for example, a subroutine that takes two reference parameters and swaps their stored values is common), this inline expansion has a significant effect on the optimization. Although this procedure-merging operation could normally be done trivially in DILS, the UCODE instruction for invoking a procedure references an external symbol-table file with an undocumented format, so I have performed the procedure merging using the DECstation utility `umerge`, which is presumably a descendent of PMERGE.

The benchmark programs were modified to run the main body of the program enough times (typically one hundred to one thousand times) to make the execution time in the tens of seconds, to ensure accurate timing and eliminate minor fluctuations due to initial loading of the cache, to which small benchmarks are susceptible. Each benchmark was then compiled and timed using each of the following four methods:

Normal The benchmark was compiled with the command `cc bench.c`. This applies the default optimization level, which is essentially the local optimization phase of UOPT.

Merged The benchmark was compiled by creating the UCODE with the default optimization level, merging the UCODE with `umerge`, and then compiling the merged UCODE with the default optimization level.

DECopt The benchmark was compiled by creating the merged UCODE, and then compiling the merged UCODE with the `-O` switch. This applies the DECstation descendent of UOPT to the merged UCODE.

Prototype The benchmark was compiled by creating the merged UCODE, applying the UOPT prototype to the merged UCODE, and then compiling the optimized UCODE with the default optimization level.

Program	Perm	Tower	Queen	Intmm	Puzzle	Quick	Bubble	Tree	Sieve
$\frac{\text{DECopt}}{\text{Normal}}$	0.94	0.53	0.90	0.45	0.47	0.53	0.54	0.72	0.50
$\frac{\text{Prototype}}{\text{Normal}}$	0.92	0.63	0.92	0.46	0.54	0.57	0.51	0.89	0.58
$\frac{\text{Merged}}{\text{Normal}}$	1.31	0.83	1.00	1.07	1.09	0.99	1.30	1.16	1.00
$\frac{\text{Prototype}}{\text{DECopt}}$	0.98	1.18	1.03	1.01	1.13	1.07	0.94	1.24	1.16

Figure 6.4: Execution time ratios

Figure 6.4 reports the ratios of the execution speeds of the variously optimized versions to the normal compilation, and the ratio of the execution speed of the code optimized by the prototype to the execution speed of the code optimized by the DECstation optimizer.

As can be seen from the figures in the table, the prototype performs comparably to the DEC UOPT optimizer. Inspection of the code produced by the optimizers reveal two major improvements in the DEC optimizer when compared to the original UOPT optimizer. One improvement involves related induction variables. Investigation of the output code indicates that the new optimizer produces better output for loops of the form

```
for i := x to y do
    A[i] := B[i] + B[i+1]
```

UOPT (and the prototype) create three induction variables for this loop, one to trace each address `A[i]`, `B[i]`, and `B[i+1]`. The DEC UOPT produces only one variable to trace both `B[i]` and `B[i+1]`; the additional offset for `B[i+1]` can folded into the load instruction. The DEC optimizer thereby reduces the overhead of initializing and keeping up-to-date one of the induction variables.

The other primary difference in the output code involves code motion. Chow's UOPT is very aggressive in performing "partial-redundancy suppression", that is, in moving code out of any control-flow structure where it may be executed more than once. If this motion goes across a procedure call, then it involves some overhead; the result of the computation needs to be stored in a memory location, and fetched back from the location when necessary. No cost analysis is performed to determine whether or not the savings in recomputation is worth the storage overhead; sometimes it is, and sometimes it isn't. This effect is very noticeable; for example, many of the "optimized" benchmarks actually run slower than the normal code when procedure merging is not performed, since the high incidence of procedure calls destroys most of the code motion benefits. The DEC UOPT appears to perform some sort of cost analysis, since it does less code motion than the prototype when procedure calls are present.

6.3.4 Efficiency of the prototype

The current implementation of Sal is inefficient, both in terms of time and space. Although some expense can be accepted in a prototyping system, it is still necessary to run experiments, and Sal is currently too slow to allow the optimization of large programs; for example, the optimization of the Sieve benchmark, a 40 line C program, took 10 minutes with the UOPT prototype.

Most of the time is currently occupied with evaluating Sal attributes. The large amount of time is due to several factors. First of all, the decision to refrain from using basic-block summary information in the prototype means that there are a huge number of nodes in the control-flow graph. Secondly, Sal uses a dynamic method to discover all attribute dependencies. Finally, the verbose encoding of type and storage information in UCODE compounds the first two problems.

Basic-block summary information was not collected in the prototype in order to ease the specification process; by making each node in the control-flow graph represent a single point in the DILS tree, the information required to propagate information through the control-flow node could be determined by inspecting the corresponding point in the tree. This eases the specification, but exacts a terrible cost in the size of the control-flow graph; for example, an arbitrarily chosen function (`fit`, from the puzzle benchmark), consisting of seven basic blocks, has six hundred fifty-eight control-flow points in the UOPT prototype's control-flow graph. There are several possible solutions to this problem. A straightforward solution is to discover basic blocks and collect summary information for them, as must be done in a production optimizer. Rather than adding this additional burden to the specification, one could constrain basic blocks to contain at most one "interesting" action, for example, one modification or use of a cell. This would eliminate the copying of unmodified attributes from one control-flow point to another, without requiring a separate representation for summary information. In the `fit` function, eliminating control-flow points that do not access the store and have only one entry and one exit reduces the number of control-flow nodes to sixty-five, an improvement by a factor of ten. At a lower level, the implementation of Sal could be modified to discover and collapse chains of dependencies that simply copy attribute values. Preliminary efforts at this modification indicate that speedups of approximately forty to fifty percent can be obtained by this fashion. Due to the dynamic nature of the dependency analysis, at least one pass over the entire control-flow graph needs to be made before this collapsing can occur; static dependency analysis combined with collapsing of copy chains might make feasible large-scale experimentation with the current UOPT prototype.

The need for dynamic dependency analysis is due to the ability to compute the node at which an attribute is being defined. In the prototypical attribute equation
`(:= (`*attribute-name node-expression*`)` *Lisp-form*`)`,
the *node-expression* can perform arbitrary computation, and so cannot be statically analyzed. But this feature is very rarely used; the *node-expression* is almost always a variable bound by pattern matching. Static dependence analysis, in those cases where it is feasible, should provide a significant speedup.

The large size of the control-flow graph is partly due to the nature of the UCODE itself. Loads and stores in UCODE contain explicit type and address information; for example, the code `(STR (LDC J 4 0) J M 18 -20 4)` is required in order to store zero into a memory location. The string of constant operands contribute ten control-flow points to the control-flow graph. These constants would not normally be present in a prototype using an IL designed in the context of Dora.

Space considerations make it impossible to compile large programs with our current machine resources (a DECstation 3100 with 24M of memory). The DILS internal representation is large; for example, the puzzle benchmark consists of approximately 25,000 bytes of symbolic UCODE, 47,000 bytes in the DILS textual representation, and 672,000 bytes in the internal representation. The internal representation size could be trimmed by careful coding. But the primary size culprit is the intermediate storage used in evaluating Sal attributes. The storage is used for closures representing attribute equations, caches for the attribute values, and tracing dependences between the caches. The number of these intermediate structures is directly dependent on the size of the control-flow graph, and so the size problem could be solved by many of the steps already suggested. The intermediate storage problem is further compounded by the method used to discover mutually recursive attribute definitions. Attributes that depend on each other must be evaluated together, and so intermediate storage for each attribute must be in use at the same time. Mutual recursion is detected dynamically in the current implementation, with the result that typically the intermediate storage for *all* attributes being evaluated is active simultaneously. A substantial savings in the intermediate space required could be effected by ordering the evaluation of attributes so that only mutually-recursive attributes are evaluated together.

Chapter 7

Conclusion

Prototyping optimizing compilers is necessary in order to make good use of the vast number of proposed optimization techniques. Dora has proven itself a useful tool in constructing prototype optimizers. In this conclusion, we examine how well Dora fulfills the desiderata put forth in the introduction, summarize the technical contributions of this dissertation, and propose future work to be done in the context of the Dora system.

7.1 Desiderata

How well does Dora fulfill the desiderata laid out for it at the beginning of its design? Briefly summarized, the desiderata stated that Dora should:

- handle a wide variety of source languages
- handle a wide variety of target machines
- be able to handle new languages and machines in a matter of days or weeks
- provide good transformation and analysis languages
- support building reusable optimizations in a language, machine, and code-level independent manner.

7.1.1 Support for different source languages

The most important influence on a compiler-building system's ability to handle a source language is the intermediate language used in the system. If the IL is too concretely tied to a set of operators incompatible with the source language, then that source language will be unusable. If the IL is too abstract, adding new source languages may involve re-writing a new compiler. As argued in Section 2.6, DILS contains the features necessary for straightforward implementation of imperative and functional languages. In general, as long as a language has

1. a notion of control-flow, that is, there is a given point in the program that executes, and then control passes to a point that is linked syntactically (for example, nearby in

the tree or linked via an identifier) or linked semantically (for example, through the store), and/or

2. a notion of data-flow, that is, operations consume the results of other operations and produce new results,

a DILS program can form a reasonable intermediate representation of a program in the language; DILS contains the most powerful control-flow and data-flow primitives currently known, and these primitives can be compiled efficiently.

There are, of course, languages that do not conform to these restrictions. As an extreme example, parser generators such as YACC [Joh75] use input languages that are heavily preprocessed and produce tables that are interpreted by fixed parsing routines. The Sal and Tess language processors likewise perform preprocessing of a relatively declarative language, doing most of their work on a non-imperative form. But compiling of such languages is currently a very language-specific art, and so reusing substantial portions of previously built compilers is not likely to be possible for these languages.

Logic languages, such as Prolog, fall somewhere between the extremes of the declarative language YACC and the obviously control-flow oriented C. Although logic languages are often termed "declarative", current logic languages have a straightforward operational semantics that uses traditional control-flow concepts. Peter Van Roy's work in Prolog compilation demonstrates that an optimizing Prolog compiler, producing code five times faster than current commercial systems, can be built that performs a substantial amount of its work on a relatively imperative intermediate form [Van90]. John Boyland is investigating the translation of Prolog programs to DILS code [Boy90].

Perhaps the biggest weakness of DILS is its lack of support for parallel programming, an area of growing importance. This lack of support stems from the relatively little work that has been done in studying important parallel primitives. The lambda expression and assignment have been around since the inception of computer science, and provide the best candidates for powerful, widely usable primitives with known implementations. Less is known regarding proper primitives for parallel processing, although Section 7.3 describes some possibilities for handling parallelism in Dora.

7.1.2 Target machine support

Little has been said in this dissertation regarding the support in Dora for generating assembly code, as Dora simply incorporates my earlier work, an extension of the instruction selection work of Pelegrí, and both of these systems are described in technical reports [Far88, Pel88]. Dora uses code generators based on tree pattern matching, and these code generators make good use of relatively standard instruction sets, so in this sense Dora can handle a wide variety of target machines. Support for parallel machines, as with parallel languages, is lacking.

7.1.3 Ease of adding new languages and machines

An ideal IL would provide excellent support for all source languages and all target machines; adding a new source language would require a simple translator from the language

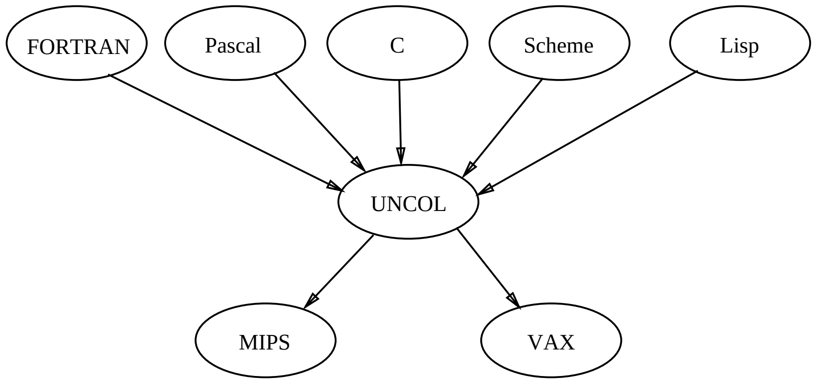


Figure 7.1: An UNCOL-based compiler for several languages and machines

to the IL, while adding a new machine would require a simple translator from the IL to the target machine language. The search for this ideal IL, or UNCOL (UNiversal Computer Oriented Language), has been going on for as long as there have been compilers; J. Strong et al., in a 1958 discussion of the UNCOL concept, state that it is “nothing particularly new” [SOW58].

The problem with the UNCOL concept is that, regardless of the intermediate language chosen, some of the source-to-IL and/or IL-to-machine translators will have to be complicated, due to the widely varied concepts present in different source languages and target machines. Consider, for example, the situation diagrammed in Figure 7.1, where various languages are compiled to a MIPS machine and to a VAX. Scheme and C have very different data types and operators; for example, Scheme must deal with dynamically typed objects, while C makes many details of the machine, such as the amount of memory occupied by various data types, available to the source language programmer. Therefore, at least one of the Scheme-UNCOL or C-UNCOL translators is likely to be a complicated undertaking. If the UNCOL does provide support for both Scheme’s and C’s operators, then the UNCOL-MIPS translator will be complicated, as it will have to translate all of the operators of two different high-level languages to machine code.

In one sense, this problem will never go away; as long as language designers continue to invent difficult-to-implement operators, compilers will have to deal with them. But Dora does make it easier to reuse existing pieces of old compilers, so that portions of languages similar to already implemented portions of other languages can be handled easily. A more realistic picture of how our multi-language/multi-target compiler might be broken down is given in Figure 7.2. Adding the first language in a new family, or the first target machine using a very different architecture, may take a considerable amount of time; but I claim that the ability to reuse old operator definitions, and to take advantage of transformations based on the properties of operators, rather than the operators themselves, means that adding similar new languages and machines should be an easy undertaking within Dora. Empirical support for this argument is yet to be established.

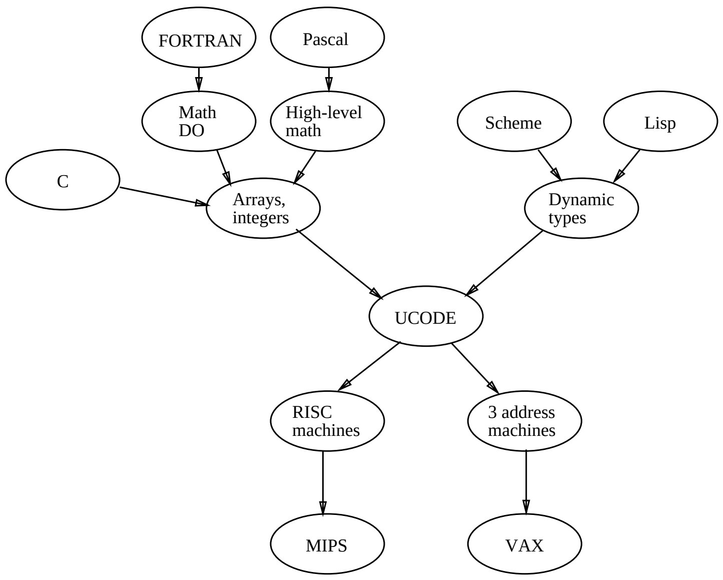


Figure 7.2: A more realistic model for a multi-language/machine compiler

7.1.4 Sal and Tess

Most of the work done to date in the Dora system has concentrated on the analysis and transformation languages. The success of Dora in these areas can be seen in the size of the code needed for the UOPT prototype; the prototype is written in about 2300 lines, compared to the 13000 lines of Pascal used in Chow's system. Of the 2300 lines, about 400 are composed of Lisp code to convert between DILS and the symbolic UCODE produced by the DEC compiler, and read and write the symbolic UCODE. Most of the remainder is composed of Sal attributes, since Chow emphasizes the use of data-flow analysis to keep the transformations simple. Sal and Tess were sufficient to implement almost all of UOPT, the exceptions being the register allocator, which uses Sal to determine live-variable spans but uses Lisp to do the actual register coloring, and a few basic block optimizations that are expressible in Sal and Tess but have simple algorithms that are as easily implemented in Lisp. As tools for describing sets of attribute equations and tree-transformations, Sal and Tess perform well; and a large number of optimizations map easily onto these two domains.

The analysis and transformation code is relatively easy to read and write. The `func-vals` attribute of Figure 6.2 is the most complicated attribute included in the prototype, and the complexity of its description is inherent in the problem it is trying to solve, rather than being an artifact of the language used to express it. The `func-vals` attribute is powerful enough to handle the full functionality of DILS, and can handle new control-flow operators that can be described by the `:returns` and `:funcalls` properties described on page 58. Adding keywords to describe more complicated control-flow operators, for example, looping operators, could be done by extending the `func-vals` attribute; in the UOPT prototype, looping operators are replaced by their implementation using the `labels` form, which is correctly analyzed by `func-vals`.

The speed of the UOPT prototype is currently too slow to allow experimentation with large programs. As discussed in Section 6.3.4, the decision to build a control-flow graph from small control-flow points instead of basic-blocks is a major contributor to this problem. Static analysis of Sal attributes, combined with collapsing of copy chains of attributes, may enable the UOPT prototype style to be used efficiently; alternatively, the prototype designer could choose to use basic-blocks for analysis.

7.1.5 Support for reusable optimizations

Finally, Dora should support writing optimizations in a reusable fashion. This aspect of Dora has also received little attention in the dissertation; more experience in writing optimizations is required before it will be clear whether this goal can be achieved. Dora supports reusable optimizations through the IL schema, which provides a uniformity of representation for different languages and machines and at different code-levels. But writing reusable optimizations still requires a large amount of parameterization. For example, eliminating common-sub-expressions is, in many ways, an ideal "reusable" optimization; the algorithms used to detect CSEs work in much the same way regardless of what language they are being applied to and whether they are being applied to source code or assembly language. But replacing a CSE with a fetch from a temporary requires the ability to allocate temporaries, and this is done quite differently at a high-level (where extra lambda expressions can be placed with abandon) than at an assembly-language level DILS (where lambda variables should already be implemented in terms of lower-level constructs). Nonetheless, even the UOPT prototype displays a measure of reusability, as the optimizations involved can be rearranged, many of them share code with each other, and portions can be easily replaced; see, for example, Grundman's experiments in replacing some of the Sal-based analysis with a language designed specifically for data-flow-analysis problems [Gru90].

In sum, Dora clearly fulfills some of the desiderata, particularly in the area of analysis and transformation languages, which has formed the central work covered in this dissertation; some of the goals for the Dora system are still open, but do not appear unattainable at this time.

7.2 Technical contributions

The major contribution of the work described in this dissertation is the Dora system itself; with Dora, it is now easier to study how well optimizations perform in practice. But two of the results of the work are important in and of themselves.

First of all, horizontal and vertical iterators form an important extension to previous work in tree pattern matching. Almost all extant systems have required the use of separately specified syntactic types, such as those declared in Dora with `define-pattern-type`, to match trees with regular structures. Feng Cheng, Scott Omdahl, and George Strawn have developed a notation that describes syntactic types inline with a pattern [COS82], but the resulting patterns are very bulky and difficult to read. Dora's iterator patterns are not only an added notational convenience; they extend the power of the tools using the pattern matching by allowing variables to be bound to repeated portions of the tree, and allowing repetitive tree portions to be instantiated using variables bound to lists.

Secondly, Sal suggests a new paradigm for defining attributes on trees. The paradigm avoids the worst problems of attribute grammars and is more expressive, as attribute grammars can be modeled easily with Sal. Sal's current drawback is a slow implementation, but I believe that this can be remedied; certainly, restricted subsets of Sal, using only pattern matching and without recursive descriptions, can be implemented efficiently.

7.3 Future work

Much work can be profitably invested in making Dora a better solution to the goals originally outlined for the system. Adding parallel constructs to Dora is a necessary step if it is to fulfill the claim of handling all common languages and machines. This can be done in Dora in two ways. First of all, parallel constructs can be built on top of DILS, using derived nodes in a manner similar to the construction of the control-flow graph. This general technique of using some higher-level "glue" to hold together a traditional sequential intermediate representation is common in compilers for parallel languages; most parallel languages have a sequential language at the core, so that the compiler's work involves additional machinery rather than a wholesale replacement. Building program-dependency graphs on top of a DILS substrate is an important step in this direction. In addition to building parallel constructs on top of DILS, new parallel primitives could be added to DILS itself. Since parallel primitives have not received nearly as much study as imperative and functional primitives, expansions in this direction would necessarily be tentative.

With DILS, Dora provides the possibility of writing reusable optimizations, but much more work needs to be done in this area to see to what extent an optimization must be parameterized to be useful in many different contexts. The operator definition language of DILS provides the opportunity to define properties of an operator that are useful for optimization. Dennis Frailey has done some early work in cataloguing such properties [Fra79], but much more needs to be done to discover just how reusable optimization code can be. In addition to information regarding the source language, target machine, and individual operators involved, many optimizations need to know something about the code level; as noted earlier, for example, removing CSEs requires the ability to allocate temporaries, a process that changes with the code level. In essence, writing a reusable optimization requires examining all of the assumptions made when implementing the optimization for a single compiler, and making the assumptions explicit in a manner usable by the system.

We have frequently noted the need for a faster implementation of Sal. Massive speed improvements could be made if more static analysis was made of the attribute definitions. Definitions that rely purely on pattern matching to set up attribute equations should be implementable using efficient techniques similar to those used in attribute grammar evaluation; incremental algorithms for these definitions also seem quite feasible, given the success of incremental attribute grammar evaluators and the incremental nature of the bottom-up pattern matching algorithms. The natural definition of some attributes, such as `func-vals`, will still require the full power of Sal, so a general evaluator is still needed; much work could be done on the implementation in its current design, including discovering copied attributes and performing straightforward source level optimization on the Sal analyzer, which is currently written using high-level Lisp features.

As Sal attributes gain in speed, it will become necessary to speed up Tess transformations as well. Currently, Tess does very little analysis of the possible rewrites at compile time. Many tree transformation systems attempt to combine as many rewrites as possible at compile time; John Boyland plans to use Tess to write many simple lowering transformations, to be used in an interactive environment as an incremental code generator, and he is investigating ways to combine several transformations into a single pass over the tree [Boy90].

Outside of the main context of Dora, Sal could be profitably used in other compiler applications. An interesting question, which I intend to pursue in the near future, is how well Sal performs as a static-semantics analysis tool when compared with attribute grammars and with the logic-constraint grammars designed by Robert Ballance [Bal89].

Finally, of course, Dora needs to be used to prototype optimizing compilers. Of primary interest is attempting to apply traditional optimizations in the UOPT mold to Lisp-oriented languages, since Kranz et al. claim that Lisp compilers are ready to take advantage of this relatively low-level optimization [KKR86].

References

[AbS85] Harold Abelson & Gerald Jay Sussman, *Structure and Interpretation of Computer Programs*, MIT Press, Cambridge, MA, 1985.

[ACF80] Frances E. Allen, J. L. Carter, J. Fabri, Jeanne Ferrante, William H. Harrison, P. G. Loewner & L. H. Trevillyan, "The Experimental Compiling System," *IBM Journal of Research and Development* 24 (1980), 695–715.

[AuH82] Marc Auslander & Martin Hopkins, "An Overview of the PL.8 Compiler," *Proceedings of the ACM-SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices* 17 (June 1982), 22–31.

[BaT86] Henri E. Bal & Andrew S. Tanenbaum, "Language- And Machine-Independent Global Optimization On Intermediate Code," *Computer Languages* 11 (1986), 105–121.

[BGV90] Robert A. Ballance, Susan L. Graham & Michael L. Van De Vanter, "The Pan Language-Based Editing System For Integrated Development Environments ," *Proceedings ACM SIGSOFT '90: Fourth Symposium on Software Development Environments* (December 1990).

[Bal89] Robert Alan Ballance, "Syntactic and Semantic Checking in Language-Based Editing Systems," Computer Science Division, EECS, Univ. of California, Berkeley, PhD Diss., Tech. Rep. 89/548, Dec. 1989.

[BaF82] Avron Barr & Edward A. Feigenbaum, eds., *The Handbook of Artificial Intelligence*, William Kaufmann, Inc., Los Altos, CA, 1982.

[Boy90] John Boyland, "Proposal for CLEAVER," internal memo, Univ. of California, Berkeley, Sept. 1990.

[Car85] Luca Cardelli, "Basic Polymorphic Typechecking," *Polymorphism* 2 (Jan. 1985).

[Cha87] David R. Chase, "An Improvement to Bottom-up Tree Pattern Matching," *Conference Record of the Fourteenth ACM Symposium on Principles of Programming Languages*, München, W. Germany (Jan. 1987), 168–177.

[COS82] Feng Cheng, Scott Omdahl & George Strawn, "Idiom matching: An optimization technique for an APL compiler," Iowa State University, Tech. Rep., Ames, IA, 1982.

[Cho83] Frederick C. Chow, "A Portable Machine-Independent Global Optimizer — Design and Measurements," Stanford Univ., PhD Diss., Dec. 1983.

[CHK86] Deborah S. Coutant, Carol L. Hammond & Jon W. Kelley, "Compilers for the New Generation of Hewlett-Packard Computers," *Hewlett-Packard J.* (Jan. 1986), 4–17.

[DJL88] Pierre Deransart, Martin Jourdan & Bernard Lorho, *Attribute Grammars*, Lect. Notes in Comp. Sci. #323, Springer Verlag, Berlin, 1988.

[Dig89] Digital Equipment Corporation, *DECstation 3100 Operator's Guide*, Maynard, Mass., Jan. 1989.

[DuC88] Gerald D. P. Dueck & Gordon V. Cormack, "Modular Attribute Grammars," Univ. of Waterloo, Tech. Rep. CS-88-19, Waterloo, Canada, May 1988.

[Far88] Charles Farnum, "A Tree Transformation Facility Using Typed Rewrite Systems," Computer Science Division, EECS, Univ. of California, Berkeley, Tech. Rep. 88/431, July 1988.

[Far82] Rodney Farrow, "Linguist 86: Yet Another Translator Writing System Based on Attribute Grammars," *Proceedings of the ACM-SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices* 17 (June 1982), 160–171.

[Far86] Rodney Farrow, "Automatic Generation of Fixed-Point-Finding Evaluators for Circular, but Well-Defined, Attribute Grammars," *Proceedings of the ACM-SIGPLAN 1986 Symposium on Compiler Construction, SIGPLAN Notices* 21 (June 1986), 85–98.

[Far89] Rodney Farrow, *The Linguist Translator-writing System — User's Manual version 6.25*, Declarative Systems Inc., Palo Alto, CA, June 1989.

[Fel87] Matthias Felleisen, "The Calculi of Lambda-v-CS Conversion: A Syntactic Theory of Control and State in Imperative Higher-order Programming Languages," Indiana University, PhD Diss., Aug. 1987.

[Fra79] Dennis J. Frailey, "An Intermediate Language for Source and Target Independent Code Optimization," *Proceedings of the ACM-SIGPLAN 1979 Symposium on Compiler Construction, SIGPLAN Notices* 14 (Aug. 1979), 188–200.

[GGM82] Harald Ganzinger, Robert Giegerich, Ulrich Moncke & Reinhard Wilhelm, "A Truly Generative Semantics Directed Compiler Generator," *Proceedings of the ACM-SIGPLAN 1982 Symposium on Compiler Construction, SIGPLAN Notices* 17 (June 1982), 172–184.

[Gru90] Douglas Grundman, "Graph Transformations and Program Flow Analysis," Computer Science Division, EECS, Univ. of California, Berkeley, PhD Diss., Dec. 1990.

[Hen84] Robert Rettig Henry, "Graham-Glanville Code Generators," Computer Science Division, EECS, Univ. of California, Berkeley, PhD Diss., Tech. Rep. 84/184, May 1984.

[His85] Andy Hisgen, "Optimization of User-Defined Types: A Program Transformation Approach," Carnegie-Mellon Univ., PhD Diss., Pittsburgh, PA, Apr. 1985.

[HoO82] Christoph M. Hoffman & Michael J. O'Donnell, "Pattern Matching in Trees," *Journal of the ACM* 29 (Jan. 1982), 68–95.

[Ing61] P. Z. Ingerman, "Thunks: a way of compiling procedure statements with some comments on procedure declarations," *Communications of the ACM* 4 (Jan. 1961), 55–58.

[Joh75] Stephen C. Johnson, "YACC: Yet Another Compiler-Compiler," Bell Laboratories, Computer Science Tech. Rep. 32, Murray Hill, NJ, July 1975.

[Jon87] Simon L. Peyton Jones, *The Implementation of Functional Programming Languages*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[KHZ82] U. Kastens, B. Hutt & E. Zimmermann, *GAG: A Practical Compiler Generator*, Lect. Notes in Comp. Sci. #141, Springer Verlag, Berlin, 1982.

[Kel89] Richard Andrews Kelsey, "Compilation by Program Transformation," Yale University, PhD Diss., May 1989.

[Ken81] Ken Kennedy, "A Survey of Data Flow Analysis Techniques," in *Program Flow Analysis: Theory and Applications*, Steven S. Muchnick & Neil D. Jones, eds., Prentice Hall, Englewood Cliffs, NJ, 1981, 5–54.

[Knu68] Donald E. Knuth, “Semantics of context free languages,” *Mathematics Systems Theory* 2 (1968), 127–145.

[KKR86] David Kranz, Richard Kelsey, Jonathan Rees, Paul Hudak, James Philbin & Norman Adams, “ORBIT: An Optimizing Compiler for Scheme,” *Proceedings of the ACM-SIGPLAN 1986 Symposium on Compiler Construction, SIGPLAN Notices* 21 (June 1986), 219–233.

[LMW88] Peter Lipps, Ulrich Möncke & Reinhard Wilhelm, “OPTRAN — A Language/System for the Specification of Program Transformations: System Overview and Experiences,” *Compiler Compilers and High Speed Compilation, 2nd CCHSC Workshop* 371 (Oct. 1988), 52–65.

[Mil78] Robin Milner, “A Theory of Type Polymorphism in Programming,” *Journal of Computer and System Sciences* 17 (1978), 348–375.

[MoR79] E. Morel & C. Renvoise, “Global Optimization by Suppression of Partial Redundancies,” *Communications of the ACM* 22 (Feb. 1979), 96–103.

[MuJ81] Steven S. Muchnick & Neil D. Jones, eds., *Program Flow Analysis: Theory and Applications*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[NoP88] Robert L. Nord & Frank Pfenning, “The Ergo Attribute System,” *Proceedings ACM SIGSOFT ’88: Third Symposium on Software Development Environments* (Nov. 1988), 110.

[Pel88] Eduardo Pelegrí-Llopart, “Rewrite Systems, Pattern Matching, and Code Generation,” Computer Science Division, EECS, Univ. of California, Berkeley, PhD Diss., Tech. Rep. 88/423, June 1988.

[Rep84] Thomas W. Reps, *Generating Language-Based Environments*, MIT Press, Cambridge, MA, 1984.

[SEF89] S. Sagiv, O. Edelstein, N. Fancez & M. Rodeh, “Resolving circularity in attribute grammars with applications to data flow analysis,” *Conference Record of the Sixteenth ACM Symposium on Principles of Programming Languages* (1989), 36.

[Sam87] A. Dain Samples, “Profile-Driven Program Improvement,” internal memo, Univ. of California, Berkeley, Mar. 1987.

[Shi88] O. Shivers, “Control Flow Analysis in Scheme,” *SIGPLAN ’88 Conference on Programming Language Design and Implementation* 23 (June 22-24, 1988), 164–174.

[Ste76] Guy Lewis Steele Jr., “LAMBDA: The Ultimate Declarative,” MIT Press, AI Memo No. 379, Cambridge, MA Artificial Intelligence Laboratories, November 1976.

[StS76] Guy Lewis Steele Jr. & Gerald Jay Sussman, “LAMBDA: The Ultimate Imperative,” MIT Press, AI Memo No. 353, Cambridge, MA Artificial Intelligence Laboratories, March 1976.

[SOW58] J. Strong, J. Olzatyn, J. Wegstein, O. Mock, A. Tritter & T. Steel, “The Problem of Programming Communications with Changing Machines,” *Communications of the ACM* 1 (Aug. 1958), 12–18.

[Tha73] James W. Thatcher, “Tree Automata: An Informal Survey,” in *Currents in the Theory of Computing*, Alfred V. Aho, ed., Prentice Hall, Englewood Cliffs, NJ, 1973, 143–172.

[Van90] Peter Van Roy, "Can logic programming execute as efficiently as imperative programming?," Computer Science Division, EECS, Univ. of California, Berkeley, PhD Diss., Dec. 1990.

[VSK89] H. H. Vogt, S. D. Swierstra & M. F. Kuiper, "Higher Order Attribute Grammars," *Proceedings of the ACM-SIGPLAN '89 Conference on Programming Language Design and Implementation* 24 (June 1989), 131–145.

[WaJ88] J. A. Walz & G. F. Johnson, "Incremental Evaluation for a General Class of Circular Attribute Grammars," *SIGPLAN '88 Conference on Programming Language Design and Implementation* 23 (June 22-24, 1988), 209–221.

[WhS90] D. Whitfield & M. L. Soffa, "An Approach to Ordering Optimizing Transformations," *Proceedings 2nd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, SIGPLAN Notices* 25 (Mar. 1990).

[Wil81] Reinhard Wilhelm, "Global Flow Analysis and Optimization in the MUG2 Compiler Generating System," in *Program Flow Analysis: Theory and Applications*, Steven S. Muchnick & Neil D. Jones, eds., Prentice Hall, Englewood Cliffs, NJ, 1981, 132–159.

Appendix A

DILS reference

DILS is an intermediate language *schema*; there is a core set of forms with a well-defined semantics, and a definition language allowing extension through the addition of new operators. The core language is based on the lambda calculus, extended with a store, continuation captures and applications, and types.

A.1 The core expressions of DILS

DILS is an expression oriented language; every construct is either an expression or an identifier. (Identifiers occur in a few well-defined places). The evaluation of an expression is dependent on a given *store*, modeling the machine memory, and an *environment*, which records bindings between variables and *cells* (units of storage). The evaluation yields a value while (possibly) modifying the store.

A.1.1 Constants

There are 3 different sets of constants available in DILS: strings, integers, and operators. When evaluated, a constant yields itself as a value and leaves the store unmodified.

The syntax of strings is identical to that used in Common Lisp. Briefly, they are surrounded by double quote marks, with double quotes escaped with a backslash:

```
"This string has a double quote here ->\"<-".
```

Integers are denoted by strings of digits, with an optional leading sign. These are the familiar mathematical integers, e.g., there are an infinite number of them.

Operators are denoted by (possibly hyphenated) alphanumeric strings. Operators must be defined before use with the `def-op` form; this form is described in Section A.2.

A.1.2 Applications

Application of a function to an argument list is denoted by enclosing the function and argument expressions in parentheses. The list of expressions (including the function expression) is evaluated left to right; the first expression must result in a function (either an operator, or a *closure*, formed by evaluating a lambda expression or label), which is then applied to the results of the argument expressions. For example, if `plus` is defined to be an operator that takes two integer arguments and returns their sum, then the expression `(plus 1 2)` evaluates to `3`.

A.1.3 Lambda expressions

A lambda expression evaluates to a "closure", a pair consisting of the lambda expression and the current environment. Lambda expressions are denoted with the syntax
`(lm (`*variables*`)` *body*`)`,
where *variables* is a sequence of zero or more variable names and *body* is an expression.

Closures can be applied as functions. Applying a closure is handled as follows:

1. The environment of the closure is extended with bindings of the variables in the lambda to new cells.

2. The new cells are initialized with the values of the actual arguments.

3. The body of the closure's lambda expression is evaluated in the extended environment; the value of the body is returned as the value of the application.

A.1.4 Variables

Lambda variables are denoted by alphanumeric names occurring in the formal parameter list of a lambda expression. Lexical scoping is used to resolve name clashes; unbound variables are not allowed (there is no implicit binding of names to cells at the global level).

Variables may be referenced in two different ways within the body of their containing lambda expression. A *cell* expression, denoted by `&`*variable*, yields the cell bound *variable* in the current environment when evaluated. A *dereference* expression, denoted `^`*variable*, yields the value currently stored in the associated cell.

A.1.5 Labels

The `labels` expression is used for mutual recursion. It takes the form

```
(labels ((label lambda-expression)
         ...)
  body)
```

The *label*s are alphanumeric names that are bound to closures formed from each corresponding *lambda-expression* and the environment of the `labels` expression. The *label*s may be used as expressions both within the body of the `labels` expression and within the lambda-expressions; a *label* evaluates to the closure to which it is bound.

A.1.6 Continuations

Consider some execution of a program just after a given function application. The application has returned a value; the remainder of the computation can be viewed as a function of the current store and the value returned. This functional abstraction of the remainder of the computation is called a *continuation.*

DILS provides two kinds of expressions dealing with continuations. Continuation captures are denoted with the form `(c-cap` *func*`)`, which applies *func* to the current continuation. Continuation applications are denoted with the form `(c-app` *cont val*`)`, which replaces the current continuation with *cont*, feeding it the value *val*. A more extensive discussion of continuations is presented in Section 2.4.

A.1.7 Core operators and types

DILS is extended through the introduction of new operator and type definitions. Types sufficient for handling the core expressions are predefined, and two operators, `assign` and `contents`, are predefined for modifying and fetching the contents of cells:

Type/Operator	Purpose
comp-type-nil	Terminator for composite type lists
fn	Type constructor for functions
label	Type constructor for labels
cont	Type constructor for continuations
void	Default type for valueless expressions, e.g., `c-caps`
integer	Type of integer constants
string	Type of string constants
cell	Type constructor for `(& v)` expressions
prim	Calling convention for operators
assign	Storage operator for cells
contents	Fetch operator for cells

The use of the type constructors is discussed in Section 2.5.

A.1.8 Global labels

Although there are no global variables, there may be global labels. Labels are globally bound to lambda expressions through the Lisp `def-label` form
`(def-label` *label type lambda-expression*`)`.
Note that this is not a DILS expression, but rather a Lisp form defining a global DILS binding. The *type* defines the type of the *lambda-expression*; types are described in Section 2.5.

A.2 Specifying DILS operators

Defining an operator requires, at the minimum, specifying some method of implementing the operator in terms of lower-level operators. Dora allows this specification to be either in terms of a DILS function definition (for example, WHILE could be implemented by a recursive function that takes two functional arguments and repeatedly invokes the second while the first returns TRUE) or by specifying a vertical transformation from a set of operators at one level to a set of operators at a lower level (for example, performing instruction selection). Operators at the bottom of the hierarchy have a one-to-one correspondence with machine language operators, and thus a trivial implementation.

Allowing operators to be defined in terms of their implementation is useful, but not sufficient; there must be a way of specifying additional information that cannot be derived from the lower-level implementation. The problem is that often a particular implementation of an operator will make irrelevant decisions that are not necessary for all legal implementations. For example, consider a simple C function[1]:

```
foo()
{
  int a;
  printf("%d\n",(int) &a);
}
```

If the "address of" function is defined in terms of low-level primitives, then this function and its surrounding program cannot be safely optimized with respect to the stack; the compiler should deduce that changing any parts of the stack allocation would change the value printed, and so changing the stack allocation would be an invalid optimization. But in this particular case, a difference in output due to a change of stack allocation is indeed acceptable. The implementor of C has to specify which parts of the likely implementation of `&` are important (the dereferencing properties of its value) and which are not (the particular integer value obtained). Thus, some specification language needs to exist to describe the important properties of operators separately from their possible implementation(s). This idea can be carried all the way out to the source language, as is done in Andy Hisgen's dissertation [His85]. Hisgen proposes a system allowing ADT implementors to specify that certain semantics-changing transformations on ADT functions are legal.

In addition to giving a method of implementing a given operator, an operator specification should also list semantic properties of the operator, particularly those that can not be derived from its implementation. For example, an operation that allocates storage space may be deleted if the space is never used; this is not derivable from its implementation, since the implementation probably has other side effects (such as changing the next storage unit to be allocated), and side effects usually cannot be ignored.

Dora provides the `def-op` form to define operators:

(def-op *name type*
  key value
  ...)

This form defines *name* as an operator with type *type*. The *key-value* pairs state additional information that can be queried by the rest of the system. Implementors may define the usage of new keywords as they see fit. Dora contains no provisions for checking the consistency of keyword usage, so keywords should be *conservative*, that is, the lack of a keyword in an operator definition should imply that no assumptions are made about that keyword rel-

[1]For the reader who is unfamiliar with C, `printf` is a library function that takes a formatting string and several additional arguments. This example defines a function `foo` with a local integer variable `a`, and prints out the address of `a`.

ative to the operator. Introducing a non-conservative keyword requires manually checking previous operator definitions.

Appendix B

Glossary for Lisp, Scheme, Sal, and Tess

Dora is implemented in Common Lisp, and the dissertation is full of examples that require some knowledge of Lisp in general and occasionally Common Lisp in particular. Details that may enable the reader unfamiliar with Lisp to comprehend the examples are described in this glossary, along with a summary of the syntax of the special forms introduced by Sal and Tess.

b/a-cps The function `b/a-cps` is defined in the UOPT prototype; it returns a list of the `before` and `after` code-points associated with a node.

Closures In any language with first-class functions, a decision has to be made regarding the treatment of free variables in the function; when the function is eventually evaluated, what values are associated with these variables? Consider the following DILS expression:

```
(labels
    ((make-simple-func (lm (num-to-return)
                           (lm () ^num-to-return)))
     (call-first (lm (x y)
                     (^x))))
  (call-first (make-simple-func 7) (make-simple-func 10)))
```

`make-simple-func` takes a number and returns a function that, when applied, returns the given number. In order for this to work as described, the storage cell bound to `num-to-return` cannot be reclaimed when `make-simple-func` is finished executing; the cell is still needed by the function returned by `make-simple-func`. This is different from the usual case in imperative languages, where formal parameters and local variables can be allocated on a stack and their space can be immediately reused when the allocating function exits. The second issue raised by the above example is that `num-to-return` is actually bound twice, the first time to a cell containing 7 and the second time to a cell containing 10. Will `call-first`, which applies its first argument, see the first binding or the second? The answer, in functional languages and in DILS, is that it sees the first; the function created by evaluating `(lm () ^num-to-return)` remembers the binding of `num-to-return` in effect at the time the lambda-expression is evaluated, rather than using the one currently in effect, or the latest in effect, at the time of the function application. This "remembering" is achieved by use of a *closure*, consisting of the body of the lambda expression (the "code") and a mapping from the free variables used by the code and their current bindings (the "environment").

cond The `cond`-expression is the Lisp equivalent of an `if...elsif...endif` expression in Algol-like languages. It has the syntax

```
(cond (bool-exp expressions)
      (bool-exp expressions)
       ...)
```

The sub-clauses of the `cond` each begin with a boolean expression; these boolean expressions are evaluated in order until a true one is found, at which point the rest of the expressions in its clause are evaluated.

CONS cells The CONS cell is a primitive data structure in Lisp, a pair of pointers used to construct linked lists and trees. Cons cells are denoted with "dotted pairs": `(`x `.` y`)` is the pair (x, y), `(`x`)` is the pair $(x,$`nil`$)$, and `(`x y `...)` is the pair $(x, (y\ \ldots))$.

contains A Sal special form, used in the definition of set attributes. `(contains` x y`)` states that the set x contains the element y.

default The clause `(default` *exp*`)` within a `def-attribute` definition provides a default expression to be evaluated for nodes that have no other attribute definition associated with them.

def-attribute The basic Sal form for defining attributes. The syntax is

```
(def-attribute attribute-name
   defining clauses).
```

`def-attribute` is described in Sections 4.3.1–4.3.4.

def-rec-attribute The Sal form for defining recursive attributes. The syntax is

```
(def-rec-attribute (name
                     :test test-func
                     :meet meet-func
                     :bottom bottom-thunk)
  one or more defining-clauses).
```

`def-rec-attribute` is described in Section 4.4.1.

def-set-attribute The Sal form for defining set attributes. The syntax is

```
(def-set-attribute name
  set-type
  one or more defining-clauses).
```

`def-set-attribute` is described in Section 4.4.1.

def-transformation The Tess form used to define transformations. Its syntax is

`(def-transformation` *name composition-rule*
  1 or more rewrite clauses`)`.

`def-transformation` is described in Section 5.2.

define-pattern-type The Dora form used to define syntactic types for pattern variables. Its syntax is

`(define-pattern-type` *name*
  1 or more patterns`)`.

`define-pattern-type` is described in Section 3.2.1.

fa-list-members A Sal/Tess special form, wrapped around defining equations or rewrite clauses. Its syntax is

`(fa-list-members (`*elt-var lisp-form*`)`
  clause`)`,

specifying that the *clause* is valid for all bindings of *elt-var* to elements of the list obtained by evaluating *lisp-form.*

fa-matches A Sal/Tess special form, wrapped around defining equations or rewrite clauses. Its syntax is

`(fa-matches (`*node-var pattern*`)`
  clause`)`,

specifying that the *clause* is valid for all bindings of *node-var* to nodes matching the *pattern.*

fa-set-members A Sal/Tess special form, wrapped around defining equations or rewrite clauses. Its syntax is

`(fa-set-members (`*elt-var lisp-form*`)`
  clause`)`,

specifying that the *clause* is valid for all bindings of *elt-var* to elements of the set obtained by evaluating *lisp-form.*

includes A Sal special form, used in the definition of set attributes. `(includes` x y`)` states that the set x includes the set y as a subset.

self In a `default` clause of a Sal attribute, `self` is bound to the node whose attribute value is being evaluated.

`->` The form `(->` *node-form output-pattern*`)` in a Tess transformation states the the node *node-form* may be replaced by a node constructed from *output-pattern.*

`:=` The form

`(:= (`*attribute-name node-expression*`)` *Lisp-form*`)`

is used within `def-attribute` to state that the attribute *attribute-name* of node *node-expression* is equal to *Lisp-form.* Both *node-expression* and *Lisp-form* are evaluated.

`>=` The form

`(>=` `(`*attribute-name node-expression*`)` *Lisp-form*`)`

is used within `def-rec-attribute` to state that the attribute *attribute-name* of node *node-expression* is greater than or equal to *Lisp-form* within the lattice. Both *node-expression* and *Lisp-form* are evaluated.

*** convention** In Common Lisp, identifiers for global constants and variables are surrounded with asterisks to distinguish them from identifiers that are locally bound.

node-set A global variable containing the set-type for sets of nodes. Set types are defined by giving an equality predicate and ordering predicate for the set elements. See also "* convention".

the-unknown-cell A constant representing the set of all storage cells that the UOPT prototype has no knowledge about. See also "* convention".

the-unknown-func A constant representing the set of all functions that the UOPT prototype has no knowledge about. See also "* convention".

tree-set A global variable containing the set-type for sets of trees. Set types are defined by giving an equality predicate and ordering predicate for the set elements. See also "* convention".

void The single DILS value of type `void`, used when an expression has no meaningful value to return. See also "* convention".

Appendix C

Sample DILS operators

This appendix contains the operator definitions used in the UOPT prototype, both those definitions used while implementing the prototype and the UCODE operators defined to optimize the benchmark programs. The intent is not to carefully describe all of the operators, but to give the reader a sense of the size of operator definitions and the difference between the two styles of IL usage.

C.1 Original operator definitions

```
;;;; Intermediate level DILS ops for Von-Neumann Languages; pascal, C, etc.

(def-type num)
(def-type ord)
(def-type real)
;;;Numeric types are represented as (num {ord|real} _type_), to help
;;;in defining generic arithmetic operators. _type_ is one of:
(def-type char)
(def-type short)
(def-type int)
(def-type long)
(def-type uchar)
(def-type ushort)
(def-type uint)
(def-type ulong)
(def-type sfloat)
(def-type float)
(def-type dfloat)
(def-type lfloat)

(def-type bool)

;;;There are two calling conventions: std, which is the system
;;;default, and inline-thunk, indicating a non-recursive parameterless
;;;lambda that can be expanded inline.
(def-convention std)
```

```
(def-convention inline-thunk)

(def-op true bool)
(def-op false bool)

;;;Making machine-type constants from DILS's mathematical integers:
(def-op make-char (fn prim (num ord char) integer))
(def-op make-short (fn prim (num ord short) integer))
(def-op make-int (fn prim (num ord int) integer))
(def-op make-long (fn prim (num ord long) integer))
(def-op make-uchar (fn prim (num ord uchar) integer))
(def-op make-ushort (fn prim (num ord ushort) integer))
(def-op make-uint (fn prim (num ord uint) integer))
(def-op make-ulong (fn prim (num ord ulong) integer))
(def-op make-sfloat (fn prim (num real sfloat) string))
(def-op make-float (fn prim (num real float) string))
(def-op make-dfloat (fn prim (num real dfloat) string))
(def-op make-lfloat (fn prim (num real lfloat) string))

;;;For conversion from one ordinal type to another, go to integer and then back:
(def-op make-integer (fn prim integer (num ord 'x)))

;;;Arrays & cell arithmetic
(def-type array)                ;(array base-type num-elts)

;;(aref some-array some-ulong) returns a cell to the some-ulong'th
;;cell of some-array.
(def-op aref (fn prim (cell 'x) (cell (array 'x 'y)) (num ord ulong)))

;;array address converts an array cell into the cell of the 1st element.
(def-op array-address (fn prim (cell 'x) (cell (array 'x 'y))))

;;cell+ and cell- have C semantics, i.e., scaling is taken into account.
(def-op cell+ (fn prim (cell 'x) (cell 'x) (num ord 'y)))
(def-op cell- (fn prim integer (cell 'x) (cell 'x)))

;;No scaling is done with address+/-.
(def-op address+ (fn prim (cell 'x) (cell 'x) (num ord 'y)))
(def-op address- (fn prim integer (cell 'x) (cell 'x)))


;;;Expression based assignment, returns the assigned value
(def-op e-assign (fn prim 'x (cell 'x) 'x)
  :stores '(0))

;;;Allocation of uninitialized storage
(def-op uninit-storage (fn prim 'res (fn 'conv 'res . 'args))
  :non-escaping-lam '(0)
  )
```

```
;;;Control flow
(def-op e1 (fn prim 'x 'x 'y)
  :unused-args '(1)
  :returns '(0)
  )

(def-op e2 (fn prim 'y 'x 'y)
  :unused-args '(0)
  :returns '(1)
  :associative t)

(def-op e-if (fn prim 'x bool (fn inline-thunk 'x) (fn inline-thunk 'x))
  :funcalls '(1 2)
  )

(def-op s-if (fn prim void bool (fn inline-thunk void) (fn inline-thunk void))
  :funcalls '(1 2)
  )

(def-op while (fn prim void (fn inline-thunk bool) (fn inline-thunk void))
  :implementation
  '(lm (test body)
    (labels ((loop (th (s-if ( ^test )
                             (th (e2 ( ^body ) (loop)))
                             (th *void*)))))
     (loop))))

(def-op repeat (fn prim void (fn inline-thunk void) (fn inline-thunk bool))
  :implementation
  '(lm (body test)
    (labels ((loop (th (e2 ( ^body )
                           (s-if ( ^test )
                                 loop
                                 (th *void*))))))
     (loop))))

;;;Arithmetic
(def-op gen- (fn prim (num 'x 'y) (num 'x 'y) (num 'x 'y)))
(def-op gen* (fn prim (num 'x 'y) (num 'x 'y) (num 'x 'y))
  :associative t)
(def-op gen+ (fn prim (num 'x 'y) (num 'x 'y) (num 'x 'y))
  :associative t)
(def-op gen/ (fn prim (num 'x 'y) (num 'x 'y) (num 'x 'y)))

(def-op and (fn prim bool bool bool))
(def-op or (fn prim bool bool bool))
(def-op xor (fn prim bool bool bool))
(def-op not (fn prim bool bool))

(def-op bitand (fn prim (num ord 'x) (num ord 'x) (num ord 'x)))
```

```
(def-op bitor (fn prim (num ord 'x) (num ord 'x) (num ord 'x)))
(def-op bitxor (fn prim (num ord 'x) (num ord 'x) (num ord 'x)))
(def-op bitnot (fn prim (num ord 'x) (num ord 'x)))

;;;Comparisons
(def-op eq (fn prim bool (num 'x 'y) (num 'x 'y)))
(def-op ne (fn prim bool (num 'x 'y) (num 'x 'y)))
(def-op lt (fn prim bool (num 'x 'y) (num 'x 'y)))
(def-op le (fn prim bool (num 'x 'y) (num 'x 'y)))
(def-op gt (fn prim bool (num 'x 'y) (num 'x 'y)))
(def-op ge (fn prim bool (num 'x 'y) (num 'x 'y)))
```

C.2 UCODE operators

```
;;;; DILS ops for UOPT

;;;UOPT encodes type information in the IL, so there are relatively
;;;few DILS types used: u-obj (something on the stack), uopt-flag (one
;;;of a small set of compile-time constants), and void.

(def-type u-obj)

(def-type uopt-flag)
;;;UOPT flags are used to indicate types and storage classes.
(def-op J uopt-flag)                    ;integer
(def-op A uopt-flag)                    ;address
(def-op L uopt-flag)                    ;long

(def-op S uopt-flag)                    ;static storage
(def-op R uopt-flag)                    ;register storage, also real data type
(def-op M uopt-flag)                    ;local proc storage
(def-op P uopt-flag)                    ;parm storage, also procedure data type

;;;UOPT is a stack-based IL.  When read internally, DILS trees are
;;;constructed by inserting the trees computing the appropriate stack
;;;elements as arguments to the operator using the elements.  The
;;;number of elements consumed is recorded as the property
;;;;:ustack-arity; the number of elements produced is one or zero,
;;;depending on whether the operator returns a u-obj or void.
(def-op LOD (fn prim u-obj uopt-flag uopt-flag integer integer integer)
  :simple-op t)                 ;load from memory to stack
(def-op ILOD (fn prim u-obj u-obj uopt-flag integer integer)
  :ustack-arity 1)              ;indirect load
(def-op STR (fn prim void u-obj uopt-flag uopt-flag integer integer integer)
  :ustack-arity 1)              ;store from stack to memory
(def-op NSTR (fn prim u-obj u-obj uopt-flag uopt-flag integer integer integer)
  :ustack-arity 1)              ;store, but do not pop
(def-op ISTR (fn prim void u-obj u-obj uopt-flag integer integer)
  :ustack-arity 2)              ;indirect store
```

```
(def-op POP (fn prim void u-obj uopt-flag)
  :ustack-arity 1)              ;pop from the stack

(def-op LDC (fn prim u-obj uopt-flag integer integer)
  :simple-op t)                 ;load a constant
(def-op MST (fn prim void integer)
  :magic-side-effects t)        ;mark stack at beginning of procedure call.
                                ;the magic-side-effects keyword indicates
                                ;side effects that don't effect any variables,
                                ;but that cannot be ignored for code-motion
                                ;purposes.
(def-op LDA (fn prim u-obj uopt-flag integer integer integer integer))
                                ;load address
(def-op PAR (fn prim void u-obj uopt-flag uopt-flag integer integer integer)
  :ustack-arity 1               ;mark top stack element as procedure parameter
  :magic-side-effects t)
(def-op CUP (fn prim void uopt-flag integer integer integer integer integer)
  :magic-side-effects t)        ;call a procedure
(def-op RET (fn prim void)
  :magic-side-effects t)        ;return from procedure

;;;Comparisons and arithmetic
(def-op LES (fn prim u-obj u-obj u-obj uopt-flag)
  :ustack-arity 2)
(def-op LEQ (fn prim u-obj u-obj u-obj uopt-flag)
  :ustack-arity 2)
(def-op GRT (fn prim u-obj u-obj u-obj uopt-flag)
  :ustack-arity 2)
(def-op GEQ (fn prim u-obj u-obj u-obj uopt-flag)
  :ustack-arity 2)
(def-op NEQ (fn prim u-obj u-obj u-obj uopt-flag)
  :ustack-arity 2)
(def-op EQU (fn prim u-obj u-obj u-obj uopt-flag)
  :ustack-arity 2)
(def-op ADD (fn prim u-obj u-obj u-obj uopt-flag)
  :ustack-arity 2)
(def-op SUB (fn prim u-obj u-obj u-obj uopt-flag)
  :ustack-arity 2)
(def-op MPY (fn prim u-obj u-obj u-obj uopt-flag)
  :ustack-arity 2)
(def-op DIV (fn prim u-obj u-obj u-obj uopt-flag)
  :ustack-arity 2)
(def-op AND (fn prim u-obj u-obj u-obj uopt-flag)
  :ustack-arity 2)
(def-op INC (fn prim u-obj u-obj uopt-flag integer)
  :ustack-arity 1)
(def-op DEC (fn prim u-obj u-obj uopt-flag integer)
  :ustack-arity 1)
(def-op CVT (fn prim u-obj u-obj uopt-flag uopt-flag)
  :ustack-arity 1)
```

```
;;;Array indexing
(def-op IXA (fn prim u-obj u-obj u-obj uopt-flag integer)
  :ustack-arity 2)

;;;Branching.  In DILS, these operators call one of their procedure
;;;arguments and then, according to standard DILS semantics, return a
;;;value.  The reading routine ensures that nothing happens after a
;;;jump, i.e., jumps only occur in tail-recursive positions.
(def-op FJP (fn prim void u-obj (fn . 'x) (fn . 'y))
  :ustack-arity 1
  :funcalls '(1 2))
(def-op TJP (fn prim void u-obj (fn . 'x) (fn . 'y))
  :ustack-arity 1
  :funcalls '(1 2))
(def-op UJP (fn prim void (fn . 'x))
  :funcalls '(0))

(def-op e2 (fn prim 'x 'y 'x)
  :unused-args '(0)
  :returns '(1)
  :associative t)

(def-op e1 (fn prim 'y 'y 'x)
  :unused-args '(1)
  :returns '(0)
  :associative t)

(def-op cse (fn prim 'x 'x)
  ;;Induction-variable replacements use this as a pseudo-op, to help in
  ;;induction-variable test replacement.  Semantically it is an
  ;;identity operator.
  )
```