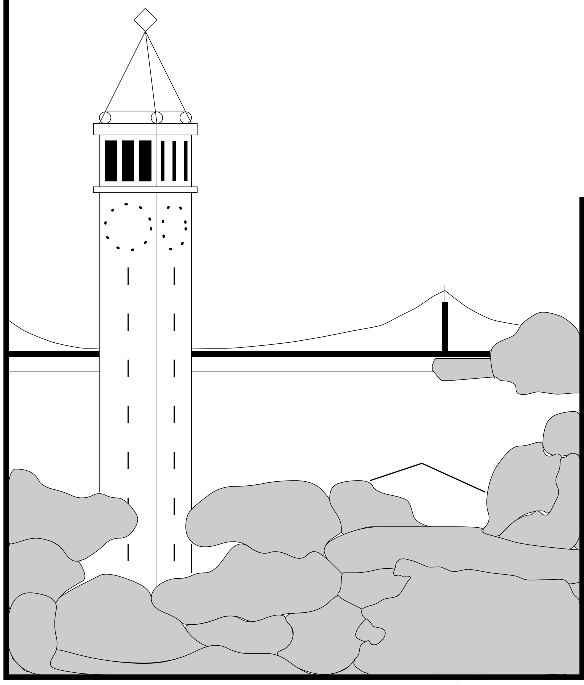
Incremental Static Semantic Analysis

William Harry Maddox III



Report No. UCB//CSD-97-948

May 1997

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Incremental Static Semantic Analysis

William Harry Maddox III

May 1997

This material is based in part upon work supported by the Advanced Research Projects Agency, Contract No. MDA972-92-J-1028. The content of the information does not necessarily reflect the position or the policy of the Government.

Abstract

Language-based programming environments provide some or all of the functionality of a compiler, an interactive debugger, a browser, and a configuration manager behind a unified user interface based on an editing paradigm. As the user edits a program, the changes are processed *incrementally*, allowing for low-latency updates to derived information. This information can be made available to interactive environment services, such as browsing, navigation, and "real time" error-reporting.

In this dissertation, we address an important subproblem in the construction of such environments, the generation of static semantic analyzers that operate in an incremental mode. Our work is embodied in the Colander II system, which introduces both a new metalanguage for the declarative specification of static semantic analyses and new techniques for generating an incremental analyzer from these specifications automatically.

Our specification metalanguage melds the advantages of traditional attribute grammars, including amenability to extensive generation-time analysis, with the expressiveness and client-independence characteristic of Ballance's Logical Constraint Grammars. In comparison to traditional attribute grammars, our metalanguage allows much more of the incrementality inherent in a particular analysis task to be exposed within the formalism itself, where it can be exploited automatically by our implementation.

Our incremental analysis algorithms exploit the attributed objects and function-valued attributes provided by our metalanguage, mapping these expressive notations onto a fine-grained incremental implementation. We are thus able to automatically generate incremental analyzers that handle long-distance dependencies and aggregate attributes efficiently. Our methods allow unusual freedom to control the granularity of incremental evaluation, allowing performance tradeoffs to be chosen as demanded by the needs of the application rather than as dictated by the *a priori* requirements of the algorithms.

We have also developed a static analysis and transformation on attribute grammars that accommodates a useful class of circular attribute dependencies, automating the "backpatching" method used in hand-coded compilers. The transformation is employed in Colander II, but is applicable to attribute grammars in general.

We have used the Colander II system to create a static semantic analyzer for the programming language Modula-2, which has revealed both strengths and weakness in our specification paradigm. Interestingly, the most significant difficulty that we encountered applies equally to traditional attribute grammars, but has not been widely appreciated in the attribute grammar literature.

Our work was performed in the context of the Ensemble project at UC Berkeley, which is constructing a prototype integrated multilingual language-based software development environment based on the editing of structured multimedia documents.

Contents

Chapter 1

Introduction

As the processing speed and memory capacity of modern computer hardware have rapidly increased, so has the demand for greater functionality in software. Despite the advances that have been made in programming languages and software development tools, programmer productivity remains a bottleneck in the software development process.

1.1 Programming Toolkits

Two decades ago, programmers typically relied on nothing more than a collection of independent programming tools, e.g., text editors, compilers, linkers, debuggers, profilers, and various utility programs. These tools were invoked explicitly by the programmer, and communicated via the filesystem using a variety of file formats and ad-hoc conventions. The first programming environments were based on a "toolkit" approach, in which mechanisms were provided to combine an extensible collection of such tools in rich and unforeseen ways to automate specific programming tasks. The most influential such environment was that provided by the Unix^{TM} operating system, whose programmable command interpreter, easy redirection of input and output, and dependency-directed compilation manager (the "`make`" utility) defined the state of the art for a decade.

A consistent failing of existing toolkit-based environments is that tools must expend much effort rederiving information that is already known to another tool, or that has not changed since the tool was last invoked. Many tools make do with an incomplete or shallow analysis in order to avoid the high cost of frequently recomputing a more extensive analysis. For example, the "tags" navigation facility in Emacs relies on the user to periodically invoke a command to update its database, which must rescan all source files that have been modified in any way. Between updates, the database may become inaccurate as the program is modified. Likewise, the language-sensitive editing modes of Emacs and other text editors rely on heuristic parsing strategies that can sometimes fail in unexpected ways, and provide no support for identifying syntax and static semantic errors, other than by invoking the compiler on the entire source file. The `make` utility records dependencies at the level of entire source files, using file timestamps maintained by the file system to detect changes, as editors neither maintain fine-grained change information, nor is the compiler able to exploit it by recompiling only the changed regions.

Ideally, we would like to have a persistent repository that is shared by all tools. In this way, the derived information created by any tool is available to any other that can make use of it. Code analysis and browsing tools such as the C Information Abstractor of Chen *et al.* [11] build a global program database that can then be queried by a programmer attempting to understand the structure of a software system. Every source file is processed once by the compiler to produce an object file, and again by the browsing tool to update its database. The information needed for the browser database is present in the symbol table of the compiler during compilation, but is not available to the the browser, thus it must reimplement a sizeable portion of the compiler's functionality. Support for browsing and analysis tools is becoming a crucial issue as the current emphasis on code re-use and re-engineering shifts the focus of the programming task away from production of new code toward the understanding, modification, and extension of existing complex systems. The information flow between the compiler and other tools need not be one-way. The global procedure call graph maintained in the browser's database could be used by the compiler for code optimization purposes as well. Ideally, both the the compiler and the browser would operate from the same persistent shared repository.

1.2 Language-Based Programming Environments

An essential limitation of loosely-coupled toolsets is that most tools know little, if anything, about the structure and semantics of the languages in which the program is expressed. Generally, only the compiler has a complete language-aware view of the program, and most of this information is discarded after each compilation is completed. True integration requires that all tools have uniform and efficient access to a semantically-rich representation of the software system. Once this infrastructure is in place, a variety of language-aware environment services can be provided in a consistent and efficient manner, without the duplication of effort that would render many of these services impractical. For example, static "compile-time" errors can be reported during editing, and incremental compilation and re-linking can be performed based on fine-grained dependencies between individual declarations, statements, and expressions, rather than between entire source files. Browsing facilities for program comprehension can consistently reflect the current status of a program even as it is undergoing modification, and editor commands for navigation and program modification can respect the structure imposed by the programming language, such as permitting consistent renaming of a variable taking the scoping of declarations into account.

Language-based programming environments provide some or all of the functionality of a compiler, interactive debugger, browser, and a configuration manager behind a unified user interface based on an editing paradigm. Changes to the program are usually processed *incrementally*, resulting in incremental updates to compile-time analyses and the generated code. These updates can usually be performed at speeds that, for interaction purposes, are essentially instantaneous. The results of the program analysis are available to support additional environment services, such as class browsing, definition/use cross-referencing, and call graph display. More extensive analyses may take longer to compute, but an integrated interactive environment can invoke such analyses speculatively during idle time (e.g., when the user is thinking) to improve the probability that a request may be satisfied without delay. A further benefit of incremental analysis is that changes can be tracked at very fine granularity, and in semantically relevant terms. For example, Wagner and Graham [77] track code revisions at the granularity of individual AST nodes, their parent-child relationships, and the annotations provided by analysis tools. Such fine-grained versioning is not provided in traditional environments, which track changes in terms of entire lines of text.

A number of language-based programming environments have been constructed as research prototypes, including Gandalf [52], Centaur [8], the Synthesizer Generator [67], Pan [3], PSG [2], and Integral-C [68]. Commercial systems include the SMARTsystem from PROCASE Corporation [60], Energize from Lucid, Inc. [51], the Rational Ada Environment [23], and a commercial version of the

Synthesizer Generator [30].[1]

Our work was performed in the context of the Ensemble project [29], which is constructing a prototype integrated multilingual language-based software development environment based on an editor for structured multimedia documents. Programs, formal specifications, and other machine-interpretable documents are represented in the the same framework as natural language documents such as user documentation, informal design notes, and management reports, with provisions for document embedding and hyperlinks. When a concrete parsing grammar is provided, e.g., for a programming language, document structure can be captured from its textual presentation. In this case, unrestricted textual editing is supported using an incremental parser. Structure-based editing is permitted in all cases. Ensemble is a successor to Pan, which pioneered the seamless treatment of programs as both flat text and as structured abstract syntax trees.

In Ensemble, a software system is represented as an abstract object. User services operate by providing graphical or textual views of that object and editing operations to modify it. Documents are represented as a tree of primitive elements and larger aggregate nodes reflecting its logical organization. In the case of a program, or other document expressed in a formal language, the structured representation takes the form of an annotated abstract syntax tree, as it is also usually represented in the other language-based environments we have cited. The abstract syntax tree is thus the shared program representation used by the services (tools) provided by these systems.

1.3 Incremental Static-semantic Analysis

In this dissertation, we address the problem of static semantic analysis in language-based programming environments, i.e., deriving properties of formal documents that can be deduced from a static examination of their context-free syntactic structure. Editing operations may modify the structure of the AST, as well as selected annotations such as user-supplied commentary. Most annotations represent the results of automatic analysis, however, and are not under direct user control. The values of these annotations are determined by the definition of the programming language and the semantics of client environment services.

Most changes that arise in editing affect a relatively small portion of the analysis. In order to preserve the information that has *not* changed, the analysis is updated incrementally as changes are made to the tree. Examples of analysis tasks are verification of context conditions, style checking, and the automatic generation of (formal) documentation.[2]

[1] There are few commercial systems, and fewer still that have been successful in the marketplace. The problem stems at least partly from the fact that most language-based environments have a closed worldview, and do not interoperate well with existing tools. This is not a technology issue *per se*, but the fact is that any single vendor cannot produce the "final word" in every tool category, so any closed architecture will leave users locked out from using tools they would like to have. In practice, a widely accepted open architecture allowing third-party "plug-ins" will be essential in order to move language-based environments into the mainstream.

[2] We are concerned principally with rather shallow analyses of the same character as those typically employed in the checking of context conditions, e.g., name-resolution and type-checking, as well as other essentially clerical tasks such as cross-referencing. We do not address most instances of deriving partial or approximate information about a program's execution-time properties, e.g., via dataflow analysis. Our techniques may possibly be generalized in this direction, but we suspect that another formalism may be more appropriate, and, in any case, our implementation methods would have to be extended. In particular, we provide only very limited support for cyclic sets of constraints, and cannot conveniently or efficiently support iterative solution methods.

1.4 Requirements

Our principal goal is to make the information conventionally derived by a compiler and related language-based tools available to other environment services, including interactive views, in a uniform and efficient manner. We must maintain this information dynamically as the program is modified, minimizing latency so that the supported environment services can provide crisp interactive response to editing operations. Several additional secondary requirements emerge from practical considerations.

Firstly, the set of language-based services will not be static, as new ones will be introduced over the lifetime of an environment. Many annotations are of general interest to both the user and environment services, including those that derive immediately from the language definition, e.g., the types of expressions and the declaration associated with each identifier use. It should be possible to add new clients without modifying existing analyses that provide the required annotations. This implies that the analysis specification must be representable without reference to its eventual clients.

Secondly, it is essential that a language-based programming environment support multiple languages, and provide for the economical addition of additional languages to its repertoire. Despite the predominance of a handful of general-purpose programming languages, large-scale software projects invariably involve a variety of languages, such as specification languages for program generators, database query languages, document markup languages, interface definition languages, and scripting languages. Mixed-language systems are becoming the rule in the commercial domain as distributed client-server designs replace the monolithic program architectures of the past. An environment that supports only a single programming language, or that cannot be easily adapted to support new languages and language variants, is unlikely to be accepted in the marketplace.

Finally, the support provided by the environment must be non-intrusive. A sequence of editing operations perceived by the user as a single modification may move the program through a sequence of intermediate inconsistent states. Since users are fully aware of the inconsistency, it is unnecessary and intrusive to notify them of the transient constraint violations that appear. We wish to support the policy, advocated by Van De Vanter [74] and adopted in Pan, of updating the analysis only when the user explicitly requests a service that depends on it, such as notification of constraint violations. Furthermore, while early language-based environments provided a structure-oriented editing interface permitting only a single subtree replacement at a time, experienced programmers demand the ability to edit the concrete textual representation of the program directly. Indeed, many modifications of an existing program that are easily performed with a conventional text editor become extremely awkward in an explicit subtree-replacement model, as noted by Waters [79]. For this reason, many language-based editors, including Ensemble, allow textual editing operations applied to a textual view of the underlying AST, and map them automatically to tree modifications with an incremental parser. The insertion or deletion of a single lexical token can cause extensive tree rearrangement, with only the final state being relevant to the semantic analyzer.

1.5 Our Solution

In this dissertation, we detail an approach that addresses all of the requirements we have identified:

- *Support for multiple languages.* We generate analyzers from high-level, declarative specifications. Our descriptions are easier to create, understand, and maintain than equivalent analyzers specified in the conventional procedural fashion. Compiled language descriptions are dynamically-loaded into the editing environment as needed. Multiple descriptions may be simultaneously active, supporting one or more documents.

- *Uniform, client-independent access.* We provide a clearly defined client interface for each analysis, textually separated from its implementation. Analysis descriptions are independent of any client.
- *Low-latency analysis.* Our evaluators execute incrementally. They are generated in the form of directly executable code, requiring minimal runtime support. Pragmatic annotations provided in the language description allow fine-grained control over space/time tradeoffs without compromising the declarative character of the description. Since some changes, even very small ones, may have wide-ranging effects, incrementality is not an adequate substitute for basic efficiency of execution. We generate hard-coded analyzers to avoid unnecessary runtime overhead.
- *Unrestricted editing model.* Our evaluators place no constraints whatsoever on either the permissible restructurings of the AST or the interleaving of tree modifications and analysis requests. In conjunction with an incremental parser, this permits any number of textual or structural changes to be made at any place in the document before requesting re-analysis. The nodes of the tree affected by a change need not be contiguous. Program transformations, whether performed automatically or by explicit user request, are easily accommodated within our framework.

Attribute grammars have been used by other researchers as a basis for incremental static semantic analysis. Unfortunately, incremental evaluators for traditional attribute grammars do not exhibit robustly scalable performance when applied to programming language analysis, leading other researchers to introduce *ad hoc* extensions that are either excessively specialized toward the description of particular programming language features, or that compromise the declarative nature of the formalism. The difficulties arise principally due to *long-distance dependencies* and *aggregate attributes*, as we shall see in the following chapter.

Our work is yet another extended attribute grammar formalism, but, unlike those proposed earlier, our extensions are not merely compromises made to facilitate incremental evaluation. Indeed, they could be justified in a non-incremental system on the grounds of their expressiveness alone. Our extensions are of broad applicability, yet they in no way compromise our ability to exploit static dependency analysis to generate efficient evaluators and to provide the strong generation-time well-formedness guarantees characteristic of systems based on traditional attribute grammars.

In many ways, our work is a carefully engineered synthesis of ideas that have been proposed or implemented before, yet there is a synergy between them that makes our system more than just the sum of these parts. We have achieved results comparable to those achieved by others, but we have done so while *enhancing* the elegance of our specification language and while retaining full faithfulness to the most desirable properties of the attribute grammar paradigm, without compromise. Perhaps just as importantly, we have done so without resorting to exceedingly clever algorithms or data structures, or by placing restrictions on other parts of the programming environment. Our approach is thus general, efficient, elegant, practical, and automated.

1.6 Summary of our Results

Our results are embodied in a system called Colander II, the analysis component of Ensemble. Colander II consists of both a new language for the formal declarative specification of analyses and new techniques for generating an incremental analyzer from these specifications.

For the analysis of languages that are a good match to our specification metalanguage, we observe significant reductions in both the number of AST node visits and in the number of attribute evaluations required to analyze a typical change during interactive editing, relative to the initial analysis when the program is first loaded. While overhead effects are less clear, as we lacked a non-incremental implementation with which to make comparisons, our methods scale well with total program size, in contrast to conventional non-incremental analysis techniques. We have used our system to create a static semantic analyzer for the programming language Modula-2, which has revealed both strengths and weakness in our specification paradigm. For the most part, our specification language was up to the task. Most of the defects we discovered could be corrected easily. Unfortunately, we discovered a more serious problem that prevented us from implementing the full semantics of Modula-2. Interestingly, the difficulty we encountered applies equally to traditional attribute grammars, but has not been widely appreciated, perhaps because an acceptable workaround is less irksome in a non-incremental setting. Most significantly, we have come to appreciate the difficulty of attempting to inherit incremental behavior in our analyzers from an incremental implementation of the semantics of the underlying metalanguage. Our generated evaluators include much redundant and unnecessary bookkeeping that would be easily avoided if coded by hand. We suggest both possible improvements to our system to close this gap, as well as an alternative approach that embodies our incremental evaluation methods in an open, lightweight framework decoupled from the monolithic "black box" specification language that forms the basis of the present work.

At present, we address programming in the small, but we eventually envision our work integrating with a persistent shared repository, extending the facilities we provide to programming in the large.

1.7 Outline of the Dissertation

Most existing language-based environments have adopted an attributed abstract syntax tree as their primary program representation, and ours is no exception. In Chapter 2, we survey existing incremental attribution systems and examine their strengths and weaknesses. This will suggest the overall character of our improved methods. We emphasize the importance of static analysis for both efficiency and reliability.

Chapter 3 presents the Analysis Definition Language (ADL), the metalanguage in which our language descriptions are expressed. This language embodies the environment builder's view of the system, and permits the specification of both language-generic and client-specific analyses. The test of our language design as a specification tool is the ease and naturalness with which analysis tasks can be expressed. In Chapter 4, we show how ADL can be used to implement typical kinds of analysis tasks, by means of an extended example.

An elegant and expressive specification language is useless for our purposes, however, unless it admits an acceptably efficient implementation. We thus describe in detail the implementation of ADL, emphasizing the novel aspects. We elaborate on the design space, defend our specific choices, and compare our approach with relevant prior work. A consistent theme is the manner in which ADL language constructs allow and encourage the environment builder to express possibilities for incremental evaluation that are concealed by other approaches, but detected and exploited by our own. In Chapter 5, we introduce classical incremental attribution techniques, and develop the particular variant we use. It is notable for its simplicity relative to previous work satisfying similar requirements. In Chapter 6, we show how our implementation of ADL objects allows the straightforward expression of long-distance attribute dependencies, and in Chapter 7 and Chapter 8 we show how function and relation-valued attributes allow fine-grained incremental evaluation in the presence of aggregate attributes. Our treatment of the incremental evaluation algorithm concludes with a unifying high-level summary and a discussion of performance issues in Chapter 9.

Classical attribute dependency analysis will reject as ill-defined any description that would create a circular data structure. This prevents a natural description of recursive data types, requiring awkward circumlocutions. Similarly, name resolution in languages that do not require declaration before use requires an awkward simulation of a multi-pass compiler structure, such as would be employed in a compiler implemented in a traditional imperative programming language. Such circumlocutions destroy the declarative elegance of the language descriptions, and create superfluous attribute values that must be maintained at additional cost. Fibering analysis, described in Chapter 10, is a static generation-time analysis that allows the noncircularity condition to be relaxed to handle these and similar situations without compromising the usual generation-time guarantee that all attributes are effectively evaluable at runtime.

Up to this point, we have spoken of our implementation in general terms, presenting general ideas and abstract algorithms. In Chapter 11 we discuss the implemented Colander II system as a concrete artifact, reporting on its implementation status and engineering details. In order to evaluate our analysis specification formalism in a realistic setting, we implemented a static semantic analyzer for the programming language Modula-2. In Chapter 12, we present an overview of the language description, which is presented in its entirety as an appendix, and report on our experience with Colander II during its construction.

Finally, in Chapter 13, we summarize our contribution and extract from our experience some conclusions and directions for future work.

Chapter 2

Formalisms for Incremental Attribution

Language-based environments conventionally represent programs as attributed abstract syntax trees. Attributes are automatically generated annotations derived from the structure of an AST. As the structure of the tree changes, the values of its attributes must be updated. It is seldom necessary to re-evaluate every attribute when an AST is modified. Methods that seek to limit the extent of re-attribution in the presence tree restructuring are called *incremental* attribution methods. In this chapter, we survey the traditional approaches to incremental attribution, then present an overview of our own.

2.1 *Ad hoc* Methods

One of the earliest syntax-directed editors, the Cornell Program Synthesizer [72], made no attempt to perform incremental analysis at all. Since it was intended for use only on small instructional programs, it was practical to perform a complete re-attribution after each editing operation. Clearly, such methods do not scale up to large modules.

In the ALOE editor [52], *action routines* are associated with each AST node type. Editing operations such as node insertion, deletion, and alteration of a child or parent pointer invoke the appropriate action routine for the node or nodes involved. The coding of such action routines is difficult and error-prone, even under the simplifying assumption of structure-oriented editing based on single subtree replacements.

While *ad hoc* attribution remains popular in batch compilers, where semantic analyzer generation tools are rarely used, nearly all later work in language-based editors abandoned *ad hoc* methods in favor of automatically generating analyzers from some sort of formal description. There is a good reason for this. Writing a correct *ad hoc* incremental analyzer is much more difficult than writing a non-incremental one. An incremental evaluation strategy must adapt to an essentially unbounded set of possible tree modifications, severely complicating the reasoning required to determine correctness.

2.2 Attribute Grammars

We can define the values of the attributes of an AST as the solution to a set of attribution equations of the form $x_0 = f(x_1, x_2, \ldots, x_N)$. Attribute grammars (AGs) provide a syntax-directed mechanism for generating such a set of equations for any program, in terms of a set of equation schemata provided for each type of AST node. The functional dependencies between the attribute instances, i.e., the variables in the equations, induce an attribute dependency graph, which we generally require to be acyclic. Given computable definitions of the functions f, we can then solve the system of equations by variable elimination. If we identify each node in the syntax tree with its associated set of attribute instances, the attribute dependency graph can be embedded in the parse tree. This is a consequence of the manner in which the system of equations is constructed by a simple tiling of instances of equation schemas provided for each node type (production rule). Attributes can thus be evaluated by walking up and down the tree, without ever explicitly constructing a dependency graph.

Attribute grammars are declarative specifications; the consistency of an attribution is defined independently of the algorithmic means by which it is achieved. For that reason, AGs are an appealing formalism for incremental analysis, in which the attribution is updated in response to unpredictable changes. AGs permit extensive generation-time dependency analysis, which facilitates the generation of efficient evaluators, and the detection of potentially circular attribute dependencies. If the semantic functions are total, the noncircularity condition guarantees that a consistent attribution must exist, and that the evaluator will be able to compute it. The user writes local attribute equations and semantic functions associated with grammar rules. The global consequences of locally apparent data dependencies are automatically determined, and the global plan of evaluation, e.g., the "pass structure" of the analyzer, can be determined automatically.

Attribute grammars embody a purely local notion of information flow, as information must be propagated node-by-node along the edges of the syntax tree. This locality is both a strength and a weakness. As a specification formalism, the compositionality of attribute grammars facilitates inductive reasoning about their meaning, which can be viewed in a completely declarative way. As a basis for language processing programs, this locality can be exploited by efficient evaluation algorithms.

Incremental attribute evaluators exploit locality in two ways. First, since the dependency information is implicit in the tree itself, it can be rederived as necessary, without maintaining the full dependency graph as a separate data structure.[1] Second, change propagation can be performed locally, propagating changes outward from the initially inconsistent attributes until no further changes are made. Incremental attribute evaluators have been designed that work in asymptotically optimal time, i.e., in $O(|\text{AFFECTED}|)$ time, where $|\text{AFFECTED}|$ is the number of attribute instances whose values must actually change.

Unfortunately, much information flow in language processing is non-syntactic, and is most naturally conceived as adding additional dependency edges, linking, for example, declarations of identifiers and their corresponding uses. These are typically "long distance" dependencies in terms of the tree path length from the declaration to the use. Once the association has been made between the declaration and its corresponding uses, it is grossly inefficient to recompute this association (by copying the binding environment attribute node-by-node) every time some attribute of the declared entity changes. Within the attribute grammar paradigm, we must accommodate such dependencies by aggregating together the information appearing at the head of such edges, distributing the aggregate to all potential use sites, and extracting the appropriate information from the aggregate at each actual use site. As the number of attributes possessed by any given node is fixed by the the text of the AG description, but the number of declared names in the program to be analyzed is unbounded, such aggregation is inevitable. Given the need for compositionality in the description, it not clear that there is any better way in general to define long-distance dependencies at the specification level, though abbreviative mechanisms may be provided as notational conveniences. For example, the *up-*

[1] Known optimal update algorithms, however, either require some explicit dependency information or restrictions on the class of acceptable attribute grammars.

ward remote attribute references introduced in the GAG system [46] permit a semantic function to refer directly to the instance of a named attribute occurring in the nearest ancestor node in which an occurrence appears. Generation-time analysis assures that such an instance will always exist, and generates inherited attributes to link the remote reference with the appropriate instance.

Long-distance dependencies lead to copy rules, i.e., equations of the form $x_0 = x_1$, which perform no computation but serve simply to allow a local tiling process to express the global dependency structure. Long chains of such copy rules can create substantial overhead in an incremental evaluator, as the set of affected attributes after the change will include all of the essentially useless intermediate attributes in the chains. When such attributes are used pervasively, e.g., for representing a binding environment that is passed to every subtree, we might have to visit every node in the program to update such an attribute. Practical evaluators must avoid this inefficiency, either by extending the attribute grammar framework, e.g., Hedin's Door Attribute Grammars [32, 33], or by using a mechanism to recognize and bypass copy rules automatically, such as those of Hoover [34] and Pennings [59].

With the elimination of copy rules, an incremental attribute evaluator can achieve fully incremental behavior to the extent that the attribute dependencies fully expose the dependency structure of the computation. For simple (e.g., scalar) attribute values, this condition obtains. However, attribute dependencies can only express dependencies directly induced by the structure of the tree. The non-syntactic dependencies are hidden away within aggregate attribute values, and are not exposed to the evaluator so as to permit incremental evaluation. The evaluator is forced to make the worst-case assumption that every reference to an aggregate depends on every component.

How significant is this in practice? Programming language analyses typically involve large, structured attribute values. The most common example is a binding environment, or "symbol table." A change to a single binding in an environment will affect only a few bindings in other environments derived from it, as well as a subset of the use sites within its scope. In a classical attribute grammar, however, if any part of such an aggregate value changes, any dependent values must be recomputed in their entirety. Thus a change to the declaration of a global variable would require that a new binding environment be propagated to *every* use of *any* variable at which the changed declaration was in scope, and force every name resolution to be performed again. This excessive overhead is experienced in every case, even when only a few use sites are actually affected by the altered declaration.

We thus see that attribute grammars, at least when implemented naively, are an inadequate specification formalism for the incremental static semantic analysis of programming languages. The value of |AFFECTED| is more an artifact of the particular form of the specification than of the degree of incrementality inherent in the analysis task, and can be artificially inflated by the constraints of the AG formalism to the point that the behavior of the "optimal-time" algorithm is entirely unacceptable. Two defects of classical attribute grammars, a crude notion of dependency and the requirement that information be propagated node-wise between adjacent nodes, conspire to conceal the underlying dependency structure of the analysis task as it might be described independently of the AG formalism. The result is evaluators that are unnecessarily slow because too many attributes must be updated in response to inessential dependencies.

The main difficulty lies in the fact that the attribute values and semantic functions in classical AGs are "black boxes," the internals of which are opaque to the evaluator generator. The computation performed by a semantic function is an atomic action, and the evaluator generator concerns itself only with the sequencing of calls to these functions and the transmission of their results to points elsewhere where they are needed. To avoid this problem, the attribution formalism must be enriched in some way that permits the AG specification to expose the possibilities for partial recomputation of aggregate attributes, and for the establishment of long-distance dependencies that can be exploited without lengthy path traversal.

The need for direct support of long-distance attribute dependencies and aggregate attributes was recognized early, and a number of rather limited or *ad hoc* methods were proposed. Beshers and Campbell [6] provide a means to integrate handwritten subsystems, e.g., a symbol table package, with an attribute grammar. The mechanism is error prone, as it relies on the user to account for the dependencies carried by the handwritten code. Hoover developed a method to automatically bypass chains of copy rules, thus automatically implementing a class of long-distance dependencies [34]. Hoover's *finite functions* [35] and *key trees* [36] are built-in associative table data types supporting efficient incremental update of their components. The semantics of these tables are "understood" by the attribute evaluator, allowing it to directly associate definitions, i.e., extensions of the mapping for a given domain value, with uses, i.e. applications of the mapping to that domain value. As an aggregation mechanism, finite functions are quite powerful, subsuming arrays, records, and keyed tables. While they appear adequate for many purposes, including name resolution in Algol-like languages, their generality is limited. Furthermore, efficient propagation of changes to the entire aggregate value relies on the ability to capture the relations between aggregate values directly in the finite function update primitive. Vorthmann and LeBlanc [76] abandon generality altogether, providing instead a specialized high-level notation for specifying name resolution rules. A similar task-specific approach is taken in PSG [2].

2.3 Enriching the AG Formalism

Colander [4], the incremental analyzer for Pan, eschewed AG technology altogether, taking an alternate approach based on incremental logic programming. Colander introduced a new attribution paradigm called *Logical Constraint Grammars* in which the AST induces not a set of equational constraints, but a set of Prolog-like goals. Goals may instantiate abstract entities and assert their properties and the relationships among them. The analysis is regarded as consistent when as many goals as possible have been satisfied. Goals that remain unsatisfied are generally the result of errors in the analyzed program, thus Colander provides a means to associate error messages with such failures. Assertions are collected in *datapools*, which are repositories of simple facts that serve as contexts for queries. Prolog-like rules allow additional facts to be deduced from those explicitly represented.

Logical Constraint Grammars rely heavily on explicit assertions, which in Prolog are an extra-logical feature that destroys fidelity to a purely declarative semantics. In Colander, however, an automatic consistency-maintenance procedure tracks the dependency of queries upon the asserted tuples examined when computing their result. When tuples are asserted or retracted (due to the subsequent failure of a goal that previously made the assertion), goals containing the affected queries are automatically retried.[2]

The tracking of dependencies among individual asserted relation tuples permits a fine-grained treatment of aggregates when modeled by relations. Unfortunately, Colander makes very little use of static analysis, and its constraint-satisfaction procedure is not sufficiently goal directed. Constraint satisfaction can block or fail to terminate due to circular dependencies, and performance suffers from

[2] The declarative semantics of Colander are rather unclean, because the constraint satisfaction process is allowed to complete with unsatisfied constraints remaining. Indeed, it is unclear exactly what the declarative semantics should be taken to be, other than the fixpoint of an idiosyncratically-defined inferencing step applied to the entire program database. The claim to declarativeness seems to derive primarily from fact that the result of an analysis must necessarily depend on the state of the current AST, independently of the sequence of editing operations that created it. In fact, even this is not precisely true due to the existence of a few operations that are exempted from the consistency-maintenance discipline.

the excessive use of dynamic dependency traces.

Horwitz [39] and Horwitz and Teitelbaum [38] augmented a traditional attribute grammar with global relations into which the nodes of the AST could induce one or more tuples, possibly conditionally. They use incremental view-maintenance techniques, akin to those developed in the relational database community, e.g., by Gupta *et al.* [31], in order to maintain additional derived relations. Their approach exploits static analysis to direct incremental re-evaluation, unlike Colander, and uses an efficient forward-inferencing procedure for updating derived relations. The relational extensions, however, are not well integrated with the traditional AG aspect, indeed they sit alongside each other, communicating but conceptually distinct. In contrast, Logical Constraint Grammars and our own attribution formalism tightly integrate extended attribute domains and their associated forms of constraint with the traditional attribution paradigm.

2.4 Our Solution: Objects, Functions, and Relations

Our attribution language provides an elegant enhancement to the modeling power of conventional AGs that avoids their failings as observed previously without giving up their distinct advantages. Component-level incrementality is exposed as a side-effect of the idiomatic use of these mechanisms, which are just as easily justified on the grounds of specification clarity as by implementation concerns. We extend the classical AG framework by introducing objects, object references, and functional and relational attributes.

An *object* is a named collection of attributes embedded within an AST node, and accessible remotely via an object reference value. A *reference to an object* is a pointer, and may be transmitted independently of the components of the object. A component may be updated in place without altering any of the reference-valued attributes that refer to the object. For the purposes of static attribute dependency analysis, an object reference is taken to depend on all of the components of the object; thus the evaluator will not schedule an access to a component before that component has been evaluated. If a component is later updated, however, the new value is visible immediately at all points in the tree where the component is accessed via an attribute reference.[3] At runtime, dependency links are maintained between an object component and the sites at which it is used, enabling the evaluator to propagate object component changes directly to the relevant locations, bypassing all of the intermediate attributes through which the object reference was transmitted. Object references establish short-cut paths for the transmission of their components, thus exposing long-distance attribute dependencies to the evaluator.

Many aggregate data structures can be encoded in the form of *functions*. A binding environment, for example, can be represented as a mapping from a name to its referent entity. Such representations are idiomatic in formal descriptions written in the denotational style, though the first-order basis of most implementations usually precludes them in attribute grammars. We permit attributes to represent functions as well as scalar values and first-order data structures such as sets or lists. Functions are not true first-class values, however, since they may not be stored as components of such data structures or passed as arguments to other functions. Functional attributes encode demand-driven tree traversals, in the sense that the evaluation of a functional attribute may depend on the functional attributes of its neighbors. Functional attributes are most commonly used as object components ("methods"). Many computations that must traverse a linked data structure can be implemented as functional attributes of the objects of which the structure is composed. This style of specification is preferable to one in which the structure to be traversed is passed explicitly to the

[3] While the value of a component may be queried remotely, the constraints that define its value always appear within the object instantiation. We do not permit remote constraints, in contrast to Boyland's formalism [10].

functions, due to the way that our implementation treats the incremental evaluation of functional attributes.

Conceptually, a function represents a possibly infinite set of argument/result tuples. Any given call, however, observes only a single tuple of the function called directly and a subset of the argument/result tuples defined by functions called indirectly. Thus, while the value of a functional attribute is conceptually its entire argument/result mapping, only a part of this mapping is relevant at any given call site. We cache the argument/result associations that are the observable values of each functional attribute. By maintaining dynamic dependency links at runtime to determine when cache entries might have become invalid, the processing of function calls whose values are known to be unaffected can be replaced by a simple retrieval of the cached value.

Logically, functions are a special case of *relations*, which generalize functions by allowing one-to-many and many-to-many associations. Relations also permit an economical representation of sets, exposing their elements for specialized incremental evaluation methods similar to those for the cached argument/result tuples of functions.[4]

From the perspective of a programmer writing a language description, our formalism has more in common with the Logical Constraint Grammars of Colander than with classical attribute grammars. Functions and relations are written using concise notations borrowed from equational logic programming. Internally, however, our implementation is strongly grounded in attribute grammar techniques, including generation-time dependency analysis. We can thus make the strong generation-time guarantees characteristic of AGs. Indeed, since our system encourages the use of small semantic functions and the avoidance of explicit recursion, and thereby exposes more of the computation to analysis, the diagnostic value of the circularity test is strengthened.

[4]The present treatment of relations is not this clever.

Chapter 3

A Metalanguage for Incremental Attribution

A key component of our solution is the metalanguage in which the static analyses are expressed. Indeed, our approach is composed in its entirety of the metalanguage itself and the methods we employ in its implementation. We call the metalanguage of the Colander II system *ADL*, which stands for *Analysis Definition Language*. Like all analysis formalisms based on attribute grammars, ADL can be considered as the combination of two components:

1. An underlying programming language that provides the data domains in terms of which the analysis results are represented, and the operators by which they are computed.

2. An attribution framework by which computations expressed in this language are associated with an abstract syntax tree.

The underlying programming language of ADL is a declarative multi-paradigm language incorporating the first-order functional, relational, and object-oriented styles of programming.

- *Functions* are defined as sets of prioritized rewrite rules of a somewhat restricted form, similar to those used in Standard ML, Haskell, and other functional programming languages favoring the "equational" style of function definition. Functions allow convenient expression of most deterministic computations.

- *Relations* are defined in the clausal style of Prolog. Relations allow the convenient expression of nondeterministic or set-oriented computations, and avoid the often unnatural asymmetry of functions in data modeling.

- *Objects* introduce the notions of identity and subtyping into the language, and play a crucial role in data modeling and in the pragmatics of incremental evaluation. Less essentially, objects provide composite data objects with named components, in contrast to the term structures characteristic of the functional and relational styles.

The functional and relational sublanguages are seamlessly integrated. They share exactly the same universe of data types, and can each easily invoke computations expressed in the other. A rewrite rule in the functional sublanguage can be guarded by a predicate in the relational sublanguage, which can both determine its applicability and bind some of its variables. A clause in the relational sublanguage can invoke a functional computation via a reducibility predicate, which succeeds, possibly binding variables, if the given expression evaluates to match a specified pattern. The object-oriented sublanguage is more restricted in scope, and consists of the addition of class types to the shared type universe.

Besides constructs of the declarative programming language, ADL analysis descriptions include a specification of the abstract syntax. Phyla, or abstract nonterminals, are declared along with a set of attributes possessed by their instances. AST operators, or abstract production rules, are provided with a set of constraints on the values of their attribute occurrences. The operation of the incremental evaluator consists mainly in assuring that the values of the attributes satisfy the applicable constraints.

In the remainder of this chapter, we present a concise and informal overview of the ADL language, highlighting its salient features and providing enough detail to allow the reader to understand the code examples in the following and subsequent chapters. Many of the finer points are omitted here, and will be introduced later as needed. A formal grammar for ADL is included as Appendix A. We conclude with a preliminary discussion of the design, illuminating our rationale behind the more controversial choices, and pointing out its connections to earlier work.

3.1 High-level Structure

An analysis description in ADL consists of two textually distinct components, the *schema*, or interface specification, and the *body*, or implementation. The schema defines the view of the analysis that is to be made accessible to its clients. Typically, the schema will provide signatures for one or more relations containing the analysis results, for example, associating error message strings with their applicable tree nodes or name usages with their corresponding declarations. Signatures for supporting functions, types, classes, and objects may also be included if needed. The body reiterates the content of the schema, providing full definitions for every exported entity, as well as elaborating the complete abstract syntax. The body may also introduce additional entities that are not visible to clients, either by including additional top-level declarations or by including additional attributes and components in its own declaration of entities that also appear in the schema.

In the remainder of this chapter, we discuss the contents of the description body. The schema is best understood as an abstraction of the body in which portions that are to be concealed from the clients are omitted. The grammar of Appendix A specifies the precise form of the declarations permitted within the schema.

3.2 Data Types

ADL is a strongly typed language in which every expression is associated with a statically determined type. The built-in primitive types include `Boolean`, `Integer`, and `String`. Values of type `Node` are references to AST nodes, i.e., the instances of AST operators. A node reference denotes a fragment of program structure or the location in the source text to which it corresponds. Node references are opaque, i.e., they do not permit access to the attributes of the node. Additional type names may be introduced as synonyms for types previously defined, for example:

```
type Count = Integer;
```

ADL supports composite types for tuples, lists, and algebraic terms. The syntax of tuple and list types mirrors that of their corresponding value constructors. For example, the tuple type representing pairs of integers and strings is denoted `(Integer, String)`, and has `(1, "foo")` as a typical value. Likewise, the list type representing a sequence of integers is denoted `[Integer]` and has `[1, 2, 3, 4, 5]` as a typical value. A list may also be constructed by prepending a sequence of components to an existing list using the "`|`" notation. For example, if the variable `Rest` has been previously bound to the list `[3, 4, 5]`, then `[1, 2 | Rest]` = `[1, 2, 3, 4, 5]`. The symbol "`|`" is part of the syntax of the list constructor, and may not be used in other contexts. Term types are similar in function to the variant records or type unions found in many other languages. An ADL term type provides a set of named constructors that can be called as functions to create values of the type. A value of a term type can be queried via pattern-matching to determine the identity of the constructor and the values of the constructor arguments that were used to create it. The following term type declaration could be used to represent the structure of a type in a simple programming language:

```
datatype TypeShape is TsUNKNOWN
                    | TsINTEGER
                    | TsPOINTER(TypeShape)
                    | TsARRAY(Integer, TypeShape)
                    ;
```

We could then represent the type of a ten-element array of pointers to integers as follows:

```
TsARRAY(10, TsPOINTER(TsINTEGER));
```

Term types may consist solely of nullary constructors, in which case they serve the purpose of enumeration types in other languages. By convention, the name of a term constructor is written in all capital letters with a short mixed-case prefix abbreviating the name of the term type to which it belongs. This convention helps avoid name conflicts between constructors belonging to different types, as well as distinguishing constructors from ordinary functions, which are conventionally written in mixed-case.

ADL supports one additional kind of type not yet introduced, the *object types*. Since the semantics of object types are closely bound to the attribution mechanism, we will defer discussion of them until we have discussed attribution.

3.3 Expressions and Patterns

Literal denotations are provided for the builtin types `Boolean`, `Integer`, and `String`:

```
TRUE     34845     "This is a string literal"
```

Names appearing in expressions may be simple identifiers, or they may be qualified identifiers with multiple components:

```
MaxInt   x   Y   Ty1   Val2   Ent.Type   VarRef.Ctx
```

An identifier appearing as an expression or subexpression must be bound to an expressible value. The names of functions and relations do not denote values, and may appear in an expression only within a call as the function or predicate to be invoked. ADL is thus essentially a first-order language, though closures may be modeled using objects, allowing some higher-order effects to be achieved.

A conventional set of operators are provided, as summarized in Figure 3.1. Equality for values of type `Node` is interpreted as node identity, while other comparisons refer to the ordering of the

Type	Operators
`Boolean`	`and`, `or`, `not` `=`, `/=` (inequality)

`Integer`	`+`, `-`, `*`, `/`, `rem`
	`-` (sign inversion)
	`=`, `/=`
	`<`, `=<`, `>`, `>=`

`String`	`^` (concatenation)
	`=`, `/=`
	`<`, `=<`, `>`, `>=`

`Node`	`=`, `/=` `<`, `=<`, `>`, `>=`
Tuples	`=`, `/=`
Lists	`^` (append) `=`, `/=`
Term Types	`=`, `/=`

Figure 3.1: ADL Operators.

nodes in a preorder traversal of the AST.[1] Equality for values of composite types is interpreted as structural isomorphism, recursively comparing all components.

A *pattern* is syntactically identical to an expression, but may contain unbound occurrences of simple (unqualified) identifiers. Such *pattern variables* must occur only as an argument to a constructor, and must not appear nested within an argument to an operator or function call. Whether a given form is to be interpreted as an expression or a pattern will always be apparent from its context. A pattern is said to *match* an expression if there exists a set of bindings for the pattern variables that would make the pattern, when evaluated, equal to that of the expression. Operationally, the successful matching of a pattern with an expression establishes bindings for the pattern variables, which may then be used subsequently as bound variables in expressions. For example, suppose that the variables **`Size`** and **`EltType`** are unbound in the current context. Then the expression

```
TsARRAY(10, TsPOINTER(TsINTEGER))
```

matches the pattern

```
TsARRAY(Size, EltType)
```

resulting in the following bindings for the previously-unbound variables:

```
Size = 10
EltType = TsPOINTER(TsINTEGER)
```

The character "`_`" in a pattern serves as a wildcard, which matches any value but establishes no bindings.

[1] While defined as part of the language for the sake of completeness, and supported by the ADL compiler itself, there is currently no runtime support for these comparisons.

Matching is a restricted form of unification in which only one operand, the pattern, is allowed to contain variables. Other than this restriction, which is statically enforced by the type system, the semantics of matching is identical to that of unification in Prolog.

3.4 Functions, Relations, and Collections

A function definition consists of a sequence of rewrite rules that provide an algorithmic specification of a mapping. The value of a function call is determined by the first rule for which the patterns of the rule head match the corresponding arguments. The bindings visible within the result expression include those established by the matching patterns.[2] For example, the factorial function might be defined as follows, where `X` is a pattern variable:

```
function Factorial(Integer) -> Integer;
  Factorial(0) => 1;
  Factorial(X) => X * Factorial(X - 1);
```

Evaluation (reduction) in ADL is performed in applicative order, i.e., ADL is a "call by value" language in which function arguments are fully evaluated prior to the call. If no rule is applicable during a function call, a runtime error is reported.

Conceptually, a relation is a set of tuples. In ADL, relations are modeled as predicates that are true of the argument values that mirror the tuples of the conceptual relation.[3] The contents of a relation are defined by a sequence of clauses:

```
relation StdFun(Integer, String);
  StdFun(1, "Sin");
  StdFun(2, "Cos");
  StdFun(3, "Tan");
```

A clause may be conditionally asserted by providing it with a *guard*. The guard may be read as a conjunction of logical predicates over a set of variables, denoting the sets of bindings of those variables for which the predicates are satisfied. The tuples contributed to the relation by each clause are those of the clause arguments evaluated within the context of each set of bindings denoted by the guard. The components of the guard, called its *literals*, may be calls to other relations or Boolean-valued expressions. The literals are separated by the "`&`" symbol, which is read "and." The predicate defined below might be used in a language description to determine names that have been given more than one definition in a scope. Let the relation `Bound` associate string names with their corresponding bound entities. Then the predicate `MultiplyDeclared` is true of those identifiers that are bound to at least two distinct entities:

```
relation MultiplyDeclared(String);
  MultiplyDeclared(Ident) :-
          Bound(Ident, Ent) &
          Bound(Ident, Other) &
          Other /= Ent;
```

Unary relations such as `MultiplyDeclared` are a convenient way to model sets; i.e., `MultiplyDeclared` may be interpreted as the set of multiply-defined identifiers.

[2]Pattern variable bindings cannot shadow pre-existing bindings, as pattern variables are, by definition, unbound in the context in which the pattern appears.

[3]We use the term "tuple" in its mathematical sense here. The tuples that compose a relation may be implicitly generated, but are not values of an ADL tuple type.

Literals may be negated by a prefix "∼" operator, and grouped using braces:

```
relation UniquelyDeclared(String);
  UniquelyDeclared(Ident) :-
          Bound(Ident, Ent) &
          ~{ Bound(Ident, Other) & Other /= Ent };
```

This predicate is true of the identifiers that are associated with exactly one entity by the relation `Bound`. Negation in ADL implements the familiar "negation as failure" rule from Prolog. Negation thus presupposes the so-called "closed-world assumption" [63] in which relations that are not specifically asserted to hold are assumed otherwise. The reducibility predicate, denoted by the infix operator `=>`, is satisfied when the expression on its left-hand side evaluates to a value matching the pattern on its right:

```
Error(Var, "Dereferenced variable must be a pointer") :-
        ~Var.Type => TsUNKNOWN &
        ~Var.Type => TsPOINTER(_);
```

This clause asserts an error at AST node `Var`, a variable reference, if the type of the variable is neither a pointer type nor the special null value `TsUNKNOWN`. The use of such null values to represent lack of information is a standard idiom in ADL, and serves to suppress spurious error messages, e.g., when an error elsewhere makes it impossible to determine a meaningful value for the attribute.

While it suffices when reading an ADL description to interpret guards solely in terms of their satisfying sets of bindings, it is necessary when writing ADL descriptions to understand their operational behavior, which motivates an important well-formedness condition. A predicate call is a generator that enumerates sets of bindings for its pattern variables. Evaluation of a guard proceeds from left to right in the same manner as the evaluation of a Prolog goal. Since ADL implements one-sided matching, however, and not full Prolog unification, evaluation of a literal necessarily results in bindings for all of its pattern variables. Each literal is thus evaluated in the context of the binding most recently established for each of the variables appearing in the literals to its left. Boolean expressions appearing in a guard cannot establish new bindings, but merely succeed or fail based on their truth value in the context of the bindings already established. This is similar to the treatment of the numeric comparison predicates in Prolog, except that the requirement that the arguments be instantiated is statically enforced in ADL.

The arguments to relation calls and the right-hand side of the reducibility operator are patterns, and may thus contain unbound pattern variables. All other arguments are expressions, in which all names must be bound, either in the scope surrounding the entire clause, or in a pattern appearing in a literal to the left. Guards that do not obey this restriction cannot be evaluated, and will be rejected during compilation of the analysis specification. Furthermore, bindings established within a negated literal are confined to that literal, and are not visible to subsequent literals. Thus a negated literal may test the bindings established by other literals to its left, but it cannot generate any new bindings of its own. This restriction is required to assure the soundness of the declarative reading of a clause as the logical conjunction of a set of predicates. It serves the same purpose as the *allowedness* condition [13] in the theory of Prolog-style logic programming.

The list inclusion predicate, denoted by the infix operator `in`, succeeds if the value of its left argument is included in that of its right, which must be a list. If the left argument is a pattern variable, i.e., a variable that has not previously been bound, the `in` operator enumerates the elements of the list as the successive bindings of the variable. More generally, the left argument may be an arbitrary pattern. In the following code fragment, the function `ArgumentMismatches` returns a list of pairs associating AST nodes with error messages. The relation `Error` represents the same associations implicitly as a relation, rather than as an explicit list:

```
Error(Locn, ErrMsg) :-
    % binding of ArgNodes, ArgTypes, and ArgSpecs elided
    ...
    ArgumentMismatches(ArgNodes, ArgTypes, ArgSpecs) => Mismatches &
    (Locn, ErrMsg) in Mismatches;
```

It is occasionally convenient to represent a relation as an explicit set of tuples, as relations are not first-class values in ADL. The `in` operator allows "impedance matching" in code fragments mixing native ADL relations and relations modeled as explicit sets of tuples.

Function rules permit an optional guard. When a guard is present, the rule is applicable not only when the argument patterns match, but the guard must be satisfied as well. The guard is evaluated within the context of the argument bindings, and may establish further bindings for the result expression. If the guard produces more than one set of bindings, a single set is chosen arbitrarily, thus preserving the deterministic semantics of function calls. This behavior is similar to the *cut* in Prolog, and may be exploited to similar effect.

```
function LocalBinding(String) -> Entity;
  LocalBinding(Ident) => Ent
      :- Binds(Ident, Ent);
  LocalBinding(Ident) => Unknown;
```

This function returns the entity bound to a given identifier as recorded in the relation `Binds`, choosing one arbitrarily if there is more than one applicable binding.

The clauses of a relation must appear within the same context as the declaration of the relation. ADL provides an alternate syntactic form of relation called a *collection*, which permits tuples to be asserted wherever the collection is accessible. A collection declared at top-level within the analysis specification is globally visible, and may have tuples asserted from within any of the AST operators. This results in considerable notational economy, as numerous intermediate relation-valued attributes threaded throughout the tree are then not required. Collections are often used to represent the global relations exported to the analysis clients. We could represent a relation associating AST nodes with error messages as follows:

```
collection Error(Node, String);
```

An AST operator might then assert an error like this:

```
Error(Addition, "Integer expression required") :-
        ~EquivTypes(Left.Type, TsINTEGER);
```

There are a number of restrictions on collections as compared to ordinary relations, motivated by their streamlined implementation and special implications for static dependency analysis. Collections may be defined only at top-level, and may be queried only by an external client. Collections are maintained in a highly efficient manner during incremental analysis, which makes their usage extremely desirable when possible.[4]

[4] The performance issues will become clear when we discuss the implementation of relations and collections in Chapter 8.

3.5 Attribution and the AST

The abstract syntax tree is composed of *nodes*, which are the instances of constructors called *operators*. Each operator belongs to a type called its *phylum*, which we also attribute to the instances of the operator. In grammatical terms, operators are the production rules of the abstract grammar and phyla are the nonterminal symbols. To avoid confusion, we will always qualify the term "operator" in this context, referring to "AST operators," reserving "operator" alone to serve in its generic sense, e.g., arithmetic operators.

A phylum declaration associates a phylum with a set of attribute signatures indicating a set of attributes possessed by every AST operator belonging to the phylum. In a simple programming language, expressions might be represented as instances of the following phylum:

```
phylum Expression
with
  context    Ctx  : Environment;
  attribute  Type : TypeShape;
where
  Type = TsUNKNOWN;
end Expression;
```

Every node belonging to the phylum `Expression` will have an attribute `Ctx` indicating the binding environment in which it appears, and an attribute `Type` indicating its type.

As in all attribute grammar formalisms, attributes are assigned a direction, either *synthesized* or *inherited*. The value of a synthesized attribute is defined in a child node and passed upward to its parent, while the value of an inherited attribute is defined in a parent node and passed downward to its children. In ADL, a simple synthesized attribute representing an expressible value is introduced using the keyword `attribute`. In a departure from conventional attribute grammars, attributes may also be functions and relations, declared by a function or relation signature of the form we have already seen, introduced by the keyword `function` or `relation`. Such attributes are always synthesized. Inherited attributes, which are restricted in ADL to represent expressible values only, are declared in the same manner as simple synthesized attributes, except the introductory keyword `context` is used in place of `attribute`.

From a purely declarative viewpoint, the inherited/synthesized distinction is not essential, as the equational constraints have no inherent directionality. Operationally, however, the distinction makes possible a practical and efficient constraint solving procedure. Equally as important, attribute directionality facilitates reasoning about the constraints, including the circularity test and related analysis performed during ADL compilation.

The equation following the keyword `where` provides a default constraint to be used if a subtree belonging to the phylum is unavailable. During template-based structure editing, for example, the AST may be only partially elaborated, or a textual change may have resulted in a parsing error that left some part of the program uninterpretable. A default constraint must be provided for each synthesized attribute.

An AST operator declaration indicates the phylum to which it belongs, the names and phyla of its children, and the constraints that apply to its attribute occurrences.

```
operator Addition : Expression is
  Left:Expression "+" Right:Expression
where
  Left.Ctx = Addition.Ctx;
```

```
    Right.Ctx = Addition.Ctx;

    Error(Addition, "Integer expression required") :-
      ~EquivTypes(Left.Type, TsINTEGER);
    Error(Addition, "Integer expression required") :-
      ~EquivTypes(Right.Type, TsINTEGER);

    Addition.Type = TsINTEGER;
  end Addition;
```

Terminal symbols enclosed in quotation marks, such as "+" above, may be included as "noise words" in order to make the declaration more readable. The parser generator used in conjunction with our system accepts the abstract grammar in similar form, requiring that the grammar so annotated actually generate the same language as the concrete grammar, though it may not be suitable for parsing. We check that the "noise words" in the ADL specification agree with those used in the grammar specification. Otherwise, they are ignored completely.

A phylum declaration may specify functions and relations as attributes. In this case, the rules or clauses take the place of attribute equations, and the syntax of function and relation names in both definitions and invocations is extended to permit qualified attribute names. The following are two trivial copy constraints, the first for a relation and the second for a function:

```
Decls.Binds(Ident, Ent) :- d.Binds(Ident, Ent);

FieldListSeq.CyclicFields(Trail) => FieldList.CyclicFields(Trail);
```

The first copies the tuples of a relational attribute `Binds` of a child node named `d` to the `Binds` attribute of the AST operator containing the clause, which is named `Decls`. The second copies the mapping defined by the functional attribute `CyclicFields` of a child named `FieldList` to the corresponding attribute of the surrounding AST operator, named `FieldListSeq`. (Although we speak of copying tuples and mappings upward, in implementation terms, each has the effect of propagating generator or function calls downward.)

While functional and relational attributes are always synthesized, we shall soon see that the effect of inherited functions and relations is easily obtained through the use of objects.

Instances of non-keyword lexemes such as identifiers and numeric literals must be explicitly represented in the AST, and can be the children of AST operators. They are thus treated similarly to AST operators, and have a syntactic type analogous to a phylum.

```
lexeme Identifier;
```

Each instance of a non-keyword lexeme has a single predefined synthesized attribute `Text`, which is its textual yield represented as a string.

In addition to the attributes specified in the phylum declaration, an AST operator may have local attributes that are declared within the body of the operator itself. Such local attributes play a role similar to local variables and local functions in conventional programming languages. Attributes may also be defined at top-level within an analysis description, outside of any AST operator. These global attributes are evaluated at the time the analysis database is initialized, and are visible everywhere within the body of the language description. Symbolic constants and utility functions are normally defined as global attributes. An abbreviated syntactic form is provided for defining a simple local or global attribute defined by a single unguarded equation:

```
attribute MaxInt  : Integer = 2147483647;
```

3.6 Object Types

We now return to object types, which were deferred in our earlier discussion of types. Unlike the other composite types of ADL, values of object types represent distinct objects possessing an identity of their own, thus allowing structurally isomorphic but separately-instantiated objects to be distinguished. The components of objects are also accessed by name instead of by positional pattern-matching.

Every object is an instance of a class, or object type, which defines a family of objects containing components with given names and signatures:

```
class VarEntity
requiring
  attribute  Type : TypeShape;
  attribute  DeclNode : Node;
end VarEntity;
```

An object is created by an *object instantiation*, which provides constraints defining the required components:

```
object VarObject : VarEntity
where
  Type = TsINTEGER;
  DeclNode = ThisOperator;
end VarObject;
```

An object instantiation is a constraint that asserts the existence of an object in much the same way that equations assert relationships among attribute values. The lifetime of the object is strictly tied to that of the AST node containing the object instantiation. Each textually-distinct object instantiation appearing in an AST operator declaration induces exactly one object instance, distinct from all others, for each instance of that operator in the AST. The component values of the object are defined by equations, rules, and clauses within the object instantiation. Within their surrounding AST operator, object components are treated as local attributes, distinguished from other local attributes only by the fact that references from outside the object instantiation must be qualified with the object name. The components of an object can be any type of attribute permitted as a local attribute, i.e., simple attributes, functions, and relations. In particular, the keyword `context` is not applicable to components, as it specifies inherited attributes.[5]

When used in an expression or pattern, the name of an object instantiation denotes a reference to the object. Likewise, when used as the type of an attribute or argument, the name of a class denotes the type of all references to objects of the class. Two references are equal when they refer to the same object. The components of an object may be accessed remotely by qualifying an object reference with the name of a component. Continuing the previous example:

```
attribute VRef : VarEntity = VarObject;

Error(Op, "Integer required") :- ~VRef.Type => TsINTEGER;
```

A class declaration may specify additional *derived* components that are computed from other components rather than specified in the object instantiation. The constraints defining these components appear within the class definition:

[5] From the standpoint of a remote component selection, in which the object reference plays a role somewhat reminiscent of a child node of an AST operator, components resemble synthesized attributes. Properly speaking, however, the inherited/synthesized distinction does not apply to components.

```
class Contour
with
  function  LocalBinding(String) -> BindingStatus;
  function  VisibleBinding(String) -> BindingStatus;
where
  LocalBinding(Ident) => Undeclared;
  VisibleBinding(Ident) => Undeclared;
end Contour;
```

A class may be declared as a subclass of another class, allowing inheritance of components. The subclass may provide additional required and derived components in addition to those inherited. A subclass may override the definition of a derived component inherited from a superclass by providing one or more new constraints. The inherited constraints are replaced, not augmented.

```
class NormalContour isa Contour
requiring
  attribute  Parent : Contour;
  relation   Binds(String, Entity);
where
  LocalBinding(Ident) => Ent
      :- Binds(Ident, Ent);
  LocalBinding(Ident) => Unknown;

  VisibleBinding(Ident) => Ent
      :- LocalBinding(Ident) => Ent & Ent /= Unknown;
  VisibleBinding(Ident) => Ent
      :- Parent.VisibleBinding(Ident) => Ent;
end NormalContour;
```

Every class is a subtype of its superclass, allowing references to objects of the class to be used in any context where a reference to an object of the superclass would be permitted. An attribute or variable may thus contain a reference to an object of a class more specific than its statically-determined type would suggest. ADL provides a special predicate, the `isa` operator, which allows a runtime test for a more specific class. As a special case, if the expression to be tested is a simple variable or an unqualified attribute name (i.e., it is syntactically an unqualified identifier), it will also be re-typed over the remainder of its scope, allowing type-safe downward coercions in the class hierarchy.

```
SimpleVar.Type = Ent.Type :-
    Ent isa VarEntity;
SimpleVar.Type = TsUNKNOWN;
```

The example above illustrates the use of *guarded equations*, in which multiple definitions are provided for an attribute, of which all but the last must be provided with a guard. The value of the attribute is determined by the first equation for which the guard is satisfied. Let `SimpleVar` be an AST operator whose instances represent simple variable references, and let `Ent` be the entity to which the name of the referenced variable is bound. The equations above define the `Type` attribute of `SimpleVar` as that of the entity, provided that it denotes a variable. Otherwise, a null value `TsUNKNOWN` is used. The selection `Ent.Type` is permitted by the ADL type-checker only because it is known that `Ent` is bound to a variable entity, which is declared to have the component `Type`, whenever the selection is performed.

3.7 Discussion

ADL is intended as a conservative design, guided by Hoare's dictum that the task of the programming language designer is "consolidation, not innovation." Our metalanguage is distinguished more by its smooth integration of familiar notions than by the presence of radically new features. The real innovation in our work lies in the manner in which ADL is implemented in order to exploit the potential for incremental execution. In an incremental evaluator, we seek to re-use work that was done in a previous analysis in order to perform the current analysis more rapidly. Unfortunately, it is not an abstract analysis that we re-use, but concrete steps in a particular computation out an infinity of possible realizations of the analysis. As we shall see in later chapters, the algorithmic form in which the analysis is specified is crucial to efficient incremental execution. What we have sought, then, is a language in which it is *natural* to express an analysis in a form suitable for incremental execution, so that efficient specifications are readable and transparent.

In our early investigations, we were greatly influenced by research in semantic data models and database programming languages. Semantic data models seek to capture the ontology of a domain as directly as possible, rather than relying on indirect "simulations" within, say, the flat tables of the traditional relational model. For example, object identity is invariably a first class notion, with "unique identifier" key values relegated to the hidden internals of an implementation. (Peckham and and Maryanski [56] provide a thorough and accessible survey of semantic data models.) Database programming languages (DBPLs) seek to eliminate the "impedance mismatch" between ordinary programming languages and embedded database query languages, and have been the subject of much research. See, for example, the volume edited by Bancilhon and Buneman [5].

As we investigated the implementation issues, however, we retreated somewhat from the ideal of a DBPL to a formalism that had a straightforward operational semantics, thus having no need for query optimization or similar technology. Our modeling facilities are closer to those provided by a typical object-oriented programming language than a DBPL based on an advanced semantic model. Nonetheless, the notion that we are modeling the static semantics, rather than just writing code, is an appealing one, and our objective is to approach this ideal as best we can within the pragmatic constraints.

Our language resembles some of the proposals that have been made to integrate predicate (clausal) logic and equational logic in logic programming, e.g., in the volume edited by Lindstrom and DeGroot[16]. We have gone much further than most of these, however, in compromising faithfulness to logical purity in favor of implementation efficiency. In particular, we enforce static restrictions on variable instantiation that preclude the familiar delayed binding of "logical variables," as well as burdening the programmer with the usual termination issues of functional programming and Prolog-style logic programming. Our evaluation engine is based on a traditional call-by-value evaluation model, extended to allow for backtracking during relational computations. Our relational computations are required to be statically moded at determinate modes, in the terminology of Reddy [62]. Indeed, all arguments of a relation are of output mode only in our present implementation. Except for collections, which support indexed access, relations blindly generate their complete set of tuples, without taking advantage of the query context at all. This often forces the use of functions or collections for performance reasons where a more general relation might seem appropriate. We nonetheless expect that our language will be quite easy to use, and that our descriptions can be *read* as declarative specifications even if operational concerns intrude into the *writing* of language descriptions. Our pragmatic approach gives us much of the practical benefit of functional logic programming while allowing the use of an efficient and well-understood evaluation model.

Some of our language design decisions are less defensible, and are simply the result of implementation expediency. We judged that single inheritance would be sufficient in our treatment of objects, even though modeling expressiveness would favor support for multiple inheritance. Indeed, for lack of multiple inheritance, the class hierarchy in our full-scale Modula-2 description is distorted slightly with respect to the one we considered most natural. The restriction of functional and relational attributes to the synthesized direction only was motivated entirely by implementation concerns, namely our direct mapping of such attributes during compilation onto methods of the target-language classes representing the AST nodes. In our limited experience, however, this restriction has been entirely without consequence.

Our criterion of "naturalness" is admittedly subjective, and it is difficult to imagine how it could be otherwise. In the next chapter, however, we will lead the reader through the implementation of a small example, and we ask him to judge for himself.

Chapter 4

Using ADL – An Example

In this chapter, we show how the constructs of ADL are used to express static semantic analyses of programs by developing an analyzer for a toy language. This language, which we call **Example**, captures the essentials of a mainstream Algol-family language, including block structure, strong typing, and declaration of named variables and types. **Example** will be used as a running example in this dissertation, illustrating both the use of the ADL language and the details of its implementation.

A program in **Example** consists of a sequence of statements. Each statement can be an assignment statement or a block that establishes one or more name bindings visible within its scope. Bindings are established by declarations, of which there are two types. Variable declarations introduce new typed storage locations. Type declarations introduce new named types. Types include integers, single-dimensional arrays, and pointers. Subscripting and dereferencing operators are provided for accessing components of structured variables. The declarations within each block may appear in any order, without any requirement that the declaration of a name textually precede uses of the name within sibling declarations.

In order to keep the volume of code to a minimum, **Example** is somewhat contrived, and admittedly would not be useful for writing real programs. In particular, there are no conditional or looping constructs, nor have we made any attempt to provide a complete family of operators. The only arithmetic operator is addition, and, while it is possible to dereference a pointer, we provide no way to properly initialize one.

The grammar of **Example** is shown in Figure 4.1. This grammar is intended to mirror the form of the abstract syntax used in the ADL language description, thus we have not attempted to remove all syntactic ambiguity. Specifically, the associativity of the operator "`+`" is not apparent, though left-associativity is intended. The grammar uses an extended BNF notation, in which the notation "{ FOO ; }+" means "one or more FOOs separated by semicolons."

4.1 Global Organization

At the outset, we must determine what information is to be derived by the analysis, and in what form it will be presented to its clients. These choices are captured in the analysis schema, shown in Figure 4.2. In a language-based environment, two important services are error reporting and content-sensitive navigation. In support of these services, we will provide two externally-visible relations for query by the environment. The relation `Error` associates nodes in the AST with the error messages that apply to them. Representing this information as a relation makes it easy to support multiple error messages for a single node. The relation `UseOf` associates the defining

program	→	*statement*
statement	→	*variable* `:=` *expression*
		`declare` *declarations* `begin` *statements* `end`
statements	→	{ *statement* `;` }+
declaration	→	`var` *id* `:` *type_spec*
		`type` *id* `=` *type_spec*
declarations	→	{ *declaration* `;` }+
type_spec	→	*id*
		`array [` *int_const* `] of` *type_spec*
		`pointer to` *type_spec*
expression	→	*variable*
		int_const
		expression `+` *expression*
variable	→	*id*
		variable `[` *expression* `]`
		variable `^`
int_const	→	{ `0-9` }+
id	→	(`a-zA-Z`){ `a-zA-Z0-9` }*

Figure 4.1: Grammar for **Example**.

```
language Example is

  relation Error(Node, String);

  relation UseOf(Node, Node);

end Example;
```

Figure 4.2: Analysis schema for **Example**.

```
language body Example : Example is

  from StringOps : StringOps import all;

  collection Error(Node, String);

  collection UseOf(Node, Node);

  ...

end Example;
```

Figure 4.3: Analysis body for **Example** (skeleton only).

occurrences of names with their uses. The programming environment may use this information to navigate from a declaration to its uses, or from a use to its corresponding declaration, or to perform a consistent renaming in a manner that respects the scoping discipline. `UseOf(X, Y)` holds whenever `Y` is the defining occurrence corresponding to the use `X`.

The body, shown in Figure 4.3, provides an implementation for each entity exported by the schema. We implement the `Error` and `UseOf` relations as collections. By special dispensation, a collection is allowed to implement a relation, as both exhibit the same semantics when queried. The `import` declaration makes available a standard library of functions on strings. The remainder of the implementation is elided here, consisting of the attribute, phylum, and AST operator definitions to be presented in the sequel.

4.2 Representing Named Entities

We will need some way to represent the two kinds of named entities provided in **Example**, simple variables and named types. Our solution is shown in Figure 4.4. Because named entities have an identity that is distinct from the properties they may possess (such as the spelling of their name), it is most appropriate to represent them as objects. The class `Entity` captures the common properties of both kinds of entities. In particular, every entity has a `DeclNode` attribute, containing the identifier node representing the defining occurrence of the name to which the entity is bound. This attribute will be examined at the use sites of the entity in order to assert tuples into the `UseOf` collection. The

```
class BindingStatus
requiring
  % nothing
end BindingStatus;

object Unknown : BindingStatus;
object Undeclared : BindingStatus;

class Entity isa BindingStatus
requiring
  attribute  DeclNode : Node;
end Entity;

class VarEntity isa Entity
requiring
  attribute  Type : TypeShape
      delayed AllVarTypes;
end VarEntity;

class TypeEntity isa Entity
requiring
  attribute  Type : TypeShape
      delayed AllTypes;
end TypeEntity;
```

Figure 4.4: Representation of named entities.

class **VarEntity** adds a **Type** attribute, representing the type of the variable. The class **TypeEntity** also adds a **Type** attribute, representing the definition of the named type. It is mere coincidence that these two attributes have the same type and are appropriately given the same name. Good modeling practice precludes simply adding a **Type** attribute to the **Entity** class. It would then be necessary to distinguish variables from type names in some other way, when this is precisely the information that is most essentially embodied in the class.

During analysis, a name resolution query may fail, either because no binding is available, or because multiple declarations of the name within the same scope render the intended referent ambiguous. Name resolution queries thus return values of class **BindingStatus**, which include two special objects, **Unknown** and **Undeclared**, representing an ambiguous referent and the absence of a binding, respectively. By deriving **Entity** as a subclass of **BindingStatus**, a query can simply return the referent entity in the normal, successful case. Alternately, we could have defined **BindingStatus** as a term type, but this would have required an additional pattern-matching step to extract the entity in some cases. With the representation chosen, we can combine the tests for successful name resolution and for the kind of entity found. The empty **requiring** clause in the declaration of class **BindingStatus** is necessary, even though the class has no attributes of its own. Without it, **BindingStatus** would be treated as a so-called *abstract class*, from which other classes may be derived, but which cannot itself be instantiated.

The keyword **delayed** is a *pragma*, i.e., a hint to the compiler. Pragmas do not affect the essential meaning of the description, but allow the programmer to control performance tradeoffs in

```
datatype TypeShape is TsUNKNOWN
                    | TsINTEGER
                    | TsPOINTER(TypeShape)
                    | TsARRAY(Integer, TypeShape)
                    | TsTYPENAME(TypeEntity)
                    ;
```

Figure 4.5: Representation of types.

its implementation or to give hints to the generation-time analysis. The function of the `delayed` pragma will be described in Section 4.8.

4.3 Representing Types

The type-matching rules of **Example** specify that a named type declaration introduces a new type distinct from any other. Otherwise, any two structurally-isomorphic types are considered equivalent. We represent the structure of a type in **Example** with an ADL value of the term type `TypeShape`, shown in Figure 4.5.

A type may belong to the builtin integer type, or it may be a pointer or an array type. A pointer type must specify the type of its referent, while an array must specify both its length and the type of its elements. A type may also be a named type, in which case the type shape includes the type name entity itself. As a final possibility, the type of a variable or expression may not be known because of an error somewhere in the program. In these cases, we use a reserved "null" type, represented by `TsUNKNOWN`. This type is treated as equivalent to any type, thus preventing confusing error messages when tested against some expected type. The use of such null values is a standard ADL idiom, adopted from conventional compiler practice.

Because type declarations may refer to other type declarations, including ones textually following, it is possible to construct cyclic named types that "contain" instances of themselves. Such recursive types may arise due to a direct reference to a type name within its own declaration, or due to a complex pattern of mutual recursion involving multiple type declarations. When mediated by pointer types, such recursive types are not problematical; in fact they are highly useful. When a type includes an instance of itself directly, however, it becomes impossible to assign a static memory allocation size to variables of the type, at least using straightforward methods without implicit pointer manipulations. For this reason, languages of the Algol-family generally forbid such recursive types, often precluding them by enforcing a declaration-before-use rule which is then relaxed slightly when a pointer type is declared. In the analysis description developed here, we test explicitly for illegal recursive types when a named type is declared. The function `CyclicType`, shown in Figure 4.6, determines if the named type given as its argument can be allocated statically.

`CyclicType` walks through the definition of the type, recursively expanding any type names, while keeping a trail of type names encountered. If a named type is found whose definition is still being expanded, then a cycle has been discovered. Since the walk does not explore pointer types, only cycles that are not mediated by pointer types are detected.

In a conventional compiler, one might implement a graph traversal of this sort by using a mark bit in each node instead of a trail. In ADL, however, no side effects such as assignments are permitted by the language. It is worth noting that all traversals of potentially cyclic structures must be

```
type TrailType = [Entity];

function TrailMember(Entity, TrailType) -> Boolean;
  TrailMember(Ent, []) => FALSE;
  TrailMember(Ent, [Ent|_]) => TRUE;
  TrailMember(Ent, [_|Rest]) => TrailMember(Ent, Rest);

function CyclicType(TypeEntity) -> Boolean;
  CyclicType(Ent) => CyclicTypeAux(Ent.Type, [Ent]);

function CyclicTypeAux(TypeShape, TrailType) -> Boolean;
  CyclicTypeAux(TsARRAY(_, ETy), Trail)
      => CyclicTypeAux(ETy, Trail);
  CyclicTypeAux(TsTYPENAME(Ent), Trail)
      => TRUE
      :- TrailMember(Ent, Trail);
  CyclicTypeAux(TsTYPENAME(Ent), Trail)
      => CyclicTypeAux(Ent.Type, [Ent|Trail]);
  CyclicTypeAux(_, _)
      => FALSE;
```

Figure 4.6: Testing for ill-formed cyclic types.

careful not to fall into an unbounded recursion. Both the `BaseType` function, which dereferences a chain of named type indirections to reveal the underlying type, and the type equivalence predicate `EquivTypes`, maintain a trail for this purpose. These functions are shown in Figure 4.7. The type equivalence rule of **Example** compares two types for structural isomorphism, with the exception that two pointer types are considered equivalent only if their referent types are denoted by the same type name. In order to prevent spurious error messages, the null type `TsUNKNOWN` is considered equivalent to any type.

4.4 Representing Binding Environments

The namespace of **Example**, like all Algol-family languages, can be thought of as a set of nested binding contours. A declaration introduces a new binding into the contour in which it appears. Every usage of a name then refers either to a binding within the immediately containing contour, or, if no such binding exists, within the innermost surrounding contour in which a binding has been declared.

In our example, we represent binding contours as objects, as shown in Figure 4.8. Every contour implements three operations, implemented as functional attributes (methods) of the contour object. The abstract class `Contour` defines this interface. The method `LocalBinding` looks up a name in the contour, and returns the relevant binding status. It looks only in the local contour, i.e., the one to which the method belongs. The method `VisibleBinding` also examines the surrounding contours, and returns the innermost binding, provided one exists. The method `Duplicate` is used only within a declaration. It is given an entity as well as a name, and returns true if any entity other than the given one is bound to the name within the local contour.

```
function BaseType(TypeShape) -> TypeShape;
  BaseType(Ty) => BaseTypeAux(Ty, []);

function BaseTypeAux(TypeShape, TrailType) -> TypeShape;
  BaseTypeAux(TsTYPENAME(Ent), Trail) => TsUNKNOWN
      :- TrailMember(Ent, Trail);
  BaseTypeAux(TsTYPENAME(Ent), Trail) => BaseTypeAux(Ent.Type, [Ent|Trail]);
  BaseTypeAux(Ty, Trail) => Ty;


function EquivTypes(TypeShape, TypeShape) -> Boolean;
  EquivTypes(Ty1, Ty2) => EquivTypesAux(Ty1, [], Ty2, []);

function EquivTypesAux(TypeShape, TrailType,
                       TypeShape, TrailType) -> Boolean;

  EquivTypesAux(TsINTEGER, Trail1, TsINTEGER, Trail2) => TRUE;
  EquivTypesAux(TsTYPENAME(Ent), Trail1, Ty, Trail2) => TRUE
      :- TrailMember(Ent, Trail1);
  EquivTypesAux(Ty, Trail1, TsTYPENAME(Ent), Trail2) => TRUE
      :- TrailMember(Ent, Trail2);
  EquivTypesAux(TsTYPENAME(Ent), Trail1, Ty, Trail2)
      => EquivTypesAux(Ent.Type, [Ent|Trail1], Ty, Trail2);
  EquivTypesAux(Ty, Trail1, TsTYPENAME(Ent), Trail2)
      => EquivTypesAux(Ty, Trail1, Ent.Type, [Ent|Trail2]);
  EquivTypesAux(TsPOINTER(TsTYPENAME(Ent)), Trail1,
                TsPOINTER(TsTYPENAME(Ent)), Trail2) => TRUE;
  EquivTypesAux(TsARRAY(Sz1, ETy1), Trail1,
                TsARRAY(Sz2, ETy2), Trail2) => TRUE
      :- Sz1 = Sz2 &
         EquivTypesAux(ETy1, Trail1, ETy2, Trail2);
  EquivTypesAux(TsUNKNOWN, Trail1, Ty, Trail2) => TRUE;
  EquivTypesAux(Ty, Trail1, TsUNKNOWN, Trail2) => TRUE;
  EquivTypesAux(TsPOINTER(TsUNKNOWN), Trail1, TsPOINTER(_), Trail2) => TRUE;
  EquivTypesAux(TsPOINTER(_), Trail1, TsPOINTER(TsUNKNOWN), Trail2) => TRUE;
  EquivTypesAux(_, _, _, _) => FALSE;
```

Figure 4.7: Testing type equivalence.

A contour normally has a parent contour and a set of bindings. The class `NormalContour` thus declares a `Parent` attribute, a reference to the object representing the surrounding contour, and a `Binds` relation, representing the bindings present locally in the contour. Both of these attributes are provided in the instantiation of the `NormalContour` object. Method overrides for the methods inherited from `Contour` implement the name lookup semantics. The search for the innermost binding performed by `VisibleBinding` is handled elegantly via delegation to the parent when needed. The outermost contour obviously cannot have a `NormalContour` as its parent, however, as all such contours must themselves have a parent. The class `NullContour` represents a trivial contour containing no bindings and having no parent. Its single instance, `NullEnv`, is used as the value of the parent attribute when instantiating the outermost contour. The synonym `Environment` is used to refer to a binding contour in its capacity as the representation of all bindings visible in a scope, including those bound in surrounding contours.

The pragma `maintained` associated with the relation `Binds` indicates that its set of tuples should be explicitly enumerated and stored, updating as needed, rather than being generated upon demand each time the relation is queried. Likewise, the pragma `maintained` associated with the function `VisibleBinding` indicates that the argument/result mappings of the function should be cached. Mappings are removed from the cache automatically when they are no longer valid, or when the function calls they represent are no longer needed to determine the current value of any attribute. Since the maintenance of caches for functions and relations involves significant overhead in both space and time, their use must be restricted to those cases that enhance the performance of incremental execution. Caching decisions must be made on the basis of an understanding of the implementation strategies employed by the ADL compiler, ideally with feedback from profiling of the language description in use.

The outermost contour contains the global environment, represented as shown in Figure 4.9, providing bindings for all predeclared names. There is only one predeclared entity in **Example**, the builtin type `Integer`.

4.5 Attributing the AST

The phyla of the abstract syntax of **Example** and their attributes are shown in Figure 4.10 and Figure 4.11.

The phylum `Program` is that of the root node of the AST, which is not permitted to have attributes. There are two non-keyword lexemes, `Id` (identifiers) and `IntConst` (integer literals). Such lexemes are treated as AST operators with a single implicitly defined attribute `Text`, representing their textual yield.

Statements, represented by the operators of phylum `Statement`, require an inherited attribute `Ctx`, representing the binding contour in which the names appearing within are to be resolved. Sequences of statements, represented by phylum `Statements`, are treated likewise. No synthesized attributes are needed.

Declarations, represented by the operators of phylum `Declaration`, require both an inherited attribute representing the binding contour in which names appearing within the declaration are to be resolved, and a synthesized attribute representing the binding that the declaration introduces. The latter is represented as a relation, `Binds`, though it will only contain one tuple. Sequences of declarations are attributed similarly, though the `Binds` relation will now contain a tuple for each declaration in the sequence. A default value must be provided for every synthesized attribute, which will be used in the event that a subtree belonging to the phylum is missing. An appropriate default value for `Binds` is the empty relation. In order to allow detection of attributes that are left

```
class Contour
with
  function  LocalBinding(String) -> BindingStatus;
  function  VisibleBinding(String) -> BindingStatus;
  function  Duplicate(String, Entity) -> Boolean;
end Contour;

class NullContour isa Contour
requiring
  % nothing
where
  LocalBinding(Ident) => Undeclared;
  VisibleBinding(Ident) => Undeclared;
  Duplicate(Ident, Ent) => FALSE;
end NullContour;

object NullEnv : NullContour ;

class NormalContour isa Contour
requiring
  attribute  Parent : Contour;
  relation   Binds(String, Entity);
where
  implement Binds as maintained;
  implement VisibleBinding as maintained;

  Duplicate(Ident, Ent) => TRUE
      :- Binds(Ident, Other) & Other /= Ent;
  Duplicate(Ident, Ent) => FALSE;

  LocalBinding(Ident) => Ent
      :- Binds(Ident, Ent) & ~Duplicate(Ident, Ent);
  LocalBinding(Ident) => Unknown;

  VisibleBinding(Ident) => Ent
      :- LocalBinding(Ident) => Ent & Ent /= Unknown;
  VisibleBinding(Ident) => Ent
      :- Parent.VisibleBinding(Ident) => Ent;
end NormalContour;

type Environment = Contour;
```

Figure 4.8: Representation of binding environments.

```
object IntType : TypeEntity
where
  Type = TsINTEGER;
  DeclNode = None;
end IntType;

object GlobalEnv : NormalContour
where
  Parent = NullEnv;
  Binds("INTEGER", IntType);
end GlobalEnv;
```

Figure 4.9: Predeclared entities.

undefined unintentionally, the ADL compiler requires that at least one constraint be provided for every attribute. In the case of the `Binds` relation, we must provide a clause whose guard always fails. The keyword `never` is a predefined synonym for `FALSE` that reads nicely in cases such as this.

Operators of phylum `TypeSpec` represent types. Names appearing within the type specification are resolved in the binding contour `Ctx`. The attribute `Type` represents the type denoted by the type specification. Operators of phyla `Expression` and `Variable` are attributed similarly, with `Type` representing the type of the expression or variable, respectively. The default value `TsUNKNOWN` is used here, in keeping with our earlier remarks regarding error recovery.

4.6 Expressions

The simplest form of expression is an integer literal, which is always of type `Integer` regardless of context. Variable references are also handled trivially, as all the real work is done in the child subtree. These are shown in Figure 4.12.

The addition operator (Figure 4.13) determines the type of its operands within the binding context in which the addition appears. Errors are asserted if the operands are not both integers. The result is always of type `Integer`.

4.7 Variables

Simple variable references are handled as shown in Figure 4.14. The current binding context is queried for the binding of the identifier `Name`. An error is asserted if no binding exists, as represented by the binding status `Undeclared`. The type of the variable reference is that of the variable entity to which the identifier is bound, provided that such a binding can be determined unambiguously. If it is bound to another kind of entity, is bound (erroneously) to multiple entities, or not bound at all, we use the null value `TsUNKNOWN` to suppress further error messages. The keyword `otherwise` is a synonym for `TRUE`. When an attribute is defined by multiple guarded equations, our preferred style includes both a final catch-all rule and the use of an explicit guard. We assert a name usage in the `UseOf` relation if any binding is found, regardless of whether it is a variable or an erroneous reference to a named type. We do not record a use if the binding is ambiguous due to erroneous multiple declarations. It could be argued that recording the variable reference as a use of *all* such

```
phylum Program;

lexeme Id;
lexeme IntConst;

phylum Statement
with
  context    Ctx : Environment;
end Statement;

phylum Statements
with
  context    Ctx : Environment;
end Statements;

phylum Declaration
with
  context    Ctx  : Environment;
  relation   Binds(String, Entity);
where
  Binds(Var, Ent) :- never;
end Declaration;

phylum Declarations
with
  context    Ctx  : Environment;
  relation   Binds(String, Entity);
where
  Binds(Var, Ent) :- never;
end Declarations;
```

Figure 4.10: Phylum declarations for **Example**.

```
phylum TypeSpec
with
  context    Ctx  : Environment;
  attribute  Type : TypeShape;
where
  Type = TsUNKNOWN;
end TypeSpec;

phylum Expression
with
  context    Ctx  : Environment;
  attribute  Type : TypeShape;
where
  Type = TsUNKNOWN;
end Expression;

phylum Variable
with
  context    Ctx  : Environment;
  attribute  Type : TypeShape;
where
  Type = TsUNKNOWN;
end Variable;
```

Figure 4.11: Phylum declarations for **Example** (continued).

```
operator ConstRef : Expression is
  Val:IntConst
where
  ConstRef.Type = TsINTEGER;
end ConstRef;


operator VarRef : Expression is
  Var:Variable
where
  Var.Ctx = VarRef.Ctx;
  VarRef.Type = Var.Type;
end VarRef;
```

Figure 4.12: Simple expressions.

```
operator Addition : Expression is
  Left:Expression "+" Right:Expression
where
  Left.Ctx = Addition.Ctx;
  Right.Ctx = Addition.Ctx;

  Error(Left, "Integer expression required") :-
    ~EquivTypes(Left.Type, TsINTEGER);
  Error(Right, "Integer expression required") :-
    ~EquivTypes(Right.Type, TsINTEGER);

  Addition.Type = TsINTEGER;
end Addition;
```

Figure 4.13: Addition operators.

declarations would be more helpful to the user, however, this would complicate the interface to the `Contour` class and add to the length of this intentionally abbreviated example.

An array variable may be subscripted by a single integer expression, as shown in Figure 4.15. Dereferencing of pointer variables is handled similarly, as shown in Figure 4.16. In both cases, the reducibility predicate (`=>`) is used to decompose the `TypeShape` terms representing the array and pointer types.

4.8 Type Specifications

A reference to a named type is handled almost exactly like a simple variable reference, differing only in that a new result type is created using the `TsTYPENAME` constructor, as shown in Figure 4.17.

It seems plausible at first sight to simply retrieve the `Type` component of the `TypeEntity` object and return it unadorned. This would permit, however, degenerate cyclic type definitions in which the definition of a named type referred directly to itself. (See Figure 4.21.) Fortunately, the ADL compiler will reject any analysis description that could exhibit this pathological behavior, as such a description would necessarily involve a circular attribute dependency. Indeed, any description that creates a circular data structure is also circular, so the reader may be wondering how we can create circular representations for recursively-defined types at all.

The ADL compiler implements a mechanism called *fibering*, which allows certain circularities involving objects to be accommodated, including the creation of cyclic data structures. Fibering relies on the observation that it is safe to schedule the evaluation of a component after the creation of a reference to its containing object, provided that every component instance is eventually evaluated prior to any attempt to access its value. When a dependency cycle is discovered in the language description, the ADL compiler attempts to eliminate it by removing the dependency of any objects within the cycle upon components for which it has been licensed to do so by a `delayed` pragma. The compiler then introduces additional scheduling constraints (dependencies) that require every instance of a component to be evaluated before any attribute instance that may access the component during its evaluation.

In Figure 4.4, the `Type` component of the `TypeEntity` class is declared as `delayed`. At evaluation time, objects of this class are initially created with the `Type` component undefined, allowing

```
operator SimpleVar : Variable is
  Name:Id
where
  attribute Ent : BindingStatus =
      SimpleVar.Ctx.VisibleBinding(Name.Text);

  Error(Name, "Undeclared variable") :- Ent = Undeclared;

  Error(Name, "Variable required") :- ~{ Ent isa VarEntity };

  UseOf(Name, Ent.DeclNode) :- Ent isa Entity;

  SimpleVar.Type = Ent.Type :-
      Ent isa VarEntity;
  SimpleVar.Type = TsUNKNOWN :-
      otherwise;
end SimpleVar;
```

Figure 4.14: Simple variable references.

```
operator SubscriptedVar : Variable is
  Var:Variable "[" Idx:Expression "]"
where
  Var.Ctx = SubscriptedVar.Ctx;
  Idx.Ctx = SubscriptedVar.Ctx;

  attribute VarTy : TypeShape = BaseType(Var.Type);

  Error(Var, "Subscripted variable must be an array") :-
      ~VarTy => TsUNKNOWN &
      ~VarTy => TsARRAY(_, _);

  attribute IdxTy : TypeShape = BaseType(Idx.Type);

  Error(Idx, "Index must be an integer expression") :-
      ~IdxTy => TsUNKNOWN &
      ~IdxTy => TsINTEGER;

  SubscriptedVar.Type = EltTy :-
      VarTy => TsARRAY(_, EltTy);
  SubscriptedVar.Type = TsUNKNOWN :-
      otherwise;
end SubscriptedVar;
```

Figure 4.15: Subscripted variables.

```
operator DereferencedVar : Variable is
  Var:Variable "^"
where
  Var.Ctx = DereferencedVar.Ctx;

  attribute VarTy : TypeShape = BaseType(Var.Type);

  Error(Var, "Dereferenced variable must be a pointer") :-
      ~VarTy => TsUNKNOWN &
      ~VarTy => TsPOINTER(_);

  DereferencedVar.Type = RefTy :-
      VarTy => TsPOINTER(RefTy);
  DereferencedVar.Type = TsUNKNOWN :-
      otherwise;
end DereferencedVar;
```

Figure 4.16: Dereferenced variables.

```
operator NamedTypeSpec : TypeSpec is
  TypeName:Id
where
  attribute Ent : BindingStatus =
      NamedTypeSpec.Ctx.VisibleBinding(TypeName.Text);

  Error(TypeName, "Undeclared type name") :- Ent = Undeclared;

  UseOf(TypeName, Ent.DeclNode) :- Ent isa Entity;

  NamedTypeSpec.Type = TsTYPENAME(Ent) :-
      Ent isa TypeEntity;
  NamedTypeSpec.Type = TsUNKNOWN :-
      otherwise;
end NamedTypeSpec;
```

Figure 4.17: Named type references.

```
operator ArrayTypeSpec : TypeSpec is
  "array" "[" Size:IntConst "]" "of" EltTy:TypeSpec
where
  EltTy.Ctx = ArrayTypeSpec.Ctx;

  ArrayTypeSpec.Type = TsARRAY(StrToInt(Size.Text), EltTy.Type);
end ArrayTypeSpec;

operator PointerTypeSpec : TypeSpec is
  "pointer" "to" RefTy:TypeSpec
where
  RefTy.Ctx = PointerTypeSpec.Ctx;

  PointerTypeSpec.Type = TsPOINTER(RefTy.Type);
end PointerTypeSpec;
```

Figure 4.18: Type constructors.

all named type definitions within a single declaration sequence to be entered into the binding environment in one pass. A second pass through the declaration sequence then completes the type definitions by computing and storing the `Type` component values omitted in the first pass. The `Type` component of the `VarDecl` class is treated similarly, as variable declarations are entered into the binding environment at the same time as the named type definitions, possibly before the definition of the declared type of the variable has been itself encountered. The effect of the fibering mechanism at evaluation time is thus similar to the use of multiple passes and "backpatching" to handle forward references in a conventional compiler.

Appropriate usage of the `delay` pragma is guided by the diagnostic dependency reports produced by the compiler, and need not be intuited *a priori*. Fibering is not always possible, as the value of a component may in fact depend on an access to the component itself. Furthermore, the static analysis upon which fibering is based is only approximate, and may sometimes fail to find a feasible evaluation schedule even when one exists. The capabilities and limitations of our fibering technique will be clarified in Chapter 10, where the fibering algorithm is presented in detail.

The constructors for array types and pointer types are straightforward, and are shown in Figure 4.18. In the case of an array type, we must obtain the length of the array as an integer. The function `StrToInt` converts a character string representing an integer into an ADL `Integer` value. This function and the auxiliary function `StrToIntAux` are defined in Figure 4.19. Two functions from the `StringOps` library are used: `StrLen` and `StrChar`. The function `StrLen(S)` returns the length of `S`. `StrChar(S, I)` returns the integer character code associated with the character at position `I` in `S`, indexing from zero.

4.9 Declarations

Variable declarations are treated as shown in Figure 4.20. An entity is instantiated to represent the variable, and a single tuple representing its binding to its name is asserted into the `Binds` relation. Named type declarations, shown in Figure 4.21, are handled in a similar fashion. A check is also made that the type is not pathologically cyclic.

```
attribute CharZero : Integer = StrChar("0", 0);

function StrToInt(String) -> Integer;
  StrToInt(Str) => StrToIntAux(Str, StrLen(Str)-1);

function StrToIntAux(String, Integer) -> Integer;
  StrToIntAux(Str, Idx) => StrChar(Str, 0) - CharZero
      :- Idx = 0;
  StrToIntAux(Str, Idx) => RestVal * 10 + StrChar(Str, Idx) - CharZero
      :- StrToIntAux(Str, Idx-1) => RestVal;
```

Figure 4.19: Converting a string to an integer.

```
operator VarDecl : Declaration is
   "var" Var:Id ":" Ty:TypeSpec
where
  Ty.Ctx = VarDecl.Ctx;

  object VarObj : VarEntity
  where
    Type = Ty.Type;
    DeclNode = Var;
  end VarObj;

  VarDecl.Binds(Var.Text, VarObj);

  Error(Var, "Multiply-declared identifier") :-
      VarDecl.Ctx.Duplicate(Var.Text, VarObj);
end VarDecl;
```

Figure 4.20: Variable declarations.

```
operator TypeDecl : Declaration is
   "type" Name:Id "=" Ty:TypeSpec
where
  Ty.Ctx = TypeDecl.Ctx;

  object TypeObj : TypeEntity
  where
    Type = Ty.Type;
    DeclNode = Name;
  end TypeObj;

  TypeDecl.Binds(Name.Text, TypeObj);

  Error(Name, "Cyclic type definition") :-
      CyclicType(TypeObj);

  Error(Name, "Multiply-declared identifier") :-
      TypeDecl.Ctx.Duplicate(Name.Text, TypeObj);
end TypeDecl;
```

Figure 4.21: Named type declarations.

In a sequence of declarations, the inherited binding environment must be distributed to each member of the sequence, and the new bindings created by the members collected into a single binding set. In AST operator `DeclList`, shown in Figure 4.22, we introduce a new ADL language construct that we have not discussed previously. The `analyze` construct allows a sequence to be attributed as if it were built up inductively from empty sequences and singletons by concatenation of subsequences. The attribute signatures following the keyword `with` define the attributes belonging to any subsequence, including empty sequences and singletons. In the present example, the attributes are the same as for the phylum `Declarations` to which the AST operator possessing the sequence child belongs. The three following `when`-clauses define an analysis by cases. The first case represents an empty sequence, providing no bindings. The second case represents a subsequence consisting of a single declaration named `d`, providing the bindings introduced by `d`. The phylum of `d` is `Declaration`, i.e., that of an element of the sequence. The final case represents a non-deterministic split of the sequence into two smaller subsequences, `d1` and `d2`, both attributed according to the `with`-clause. The bindings provided by this case are the union of those provided by each component subsequence.

This treatment of sequence attribution in ADL allows for flexibility in the implementation of sequences, which are represented internally as clusters of fixed-arity tree nodes. Alternate representations more commonly used, particularly when sequences are not provided as primitives in the syntactic metalanguage, commit to either a left-recursive or a right-recursive decomposition. An attribution based on such an asymmetric decomposition cannot be translated straightforwardly into a form suitable for use with an underlying node structure of another form. While the `analyze` construct could be used to attribute a sequence represented in a linear-recursive fashion, it is more naturally adapted to a symmetric representation that mirrors the threefold case analysis on which the abstract view of the sequence is based. In the next chapter, we explain how a symmetric representation is better suited to incremental evaluation than an asymmetric one.

Each case within `analyze` is effectively an AST operator definition, and may contain any of

```
operator DeclList : Declarations is
  { Decls:Declaration ";" }*
where
  Decls.Ctx = DeclList.Ctx;

  analyze Decls
  with
    context    Ctx  : Environment;
    relation   Binds(String, Entity);
  when [] =>
    % empty sequence
    Decls.Binds(Ident, Ent) :- never;
  when [ d ] =>
    % singleton sequence
    anchor;
    d.Ctx = Decls.Ctx;
    Decls.Binds(Ident, Ent) :- d.Binds(Ident, Ent);
  when [ d1 ^ d2 ] =>
    % nondeterministic sequence split
    anchor;
    d1.Ctx = Decls.Ctx;
    d2.Ctx = Decls.Ctx;
    Decls.Binds(Ident, Ent) :- d1.Binds(Ident, Ent);
    Decls.Binds(Ident, Ent) :- d2.Binds(Ident, Ent);
  end;

  DeclList.Binds(Ident, Ent) :- Decls.Binds(Ident, Ent);
end DeclList;
```

Figure 4.22: Declaration sequences.

```
operator Assignment : Statement is
  Var:Variable ":=" Val:Expression
where
  Var.Ctx = Assignment.Ctx;
  Val.Ctx = Assignment.Ctx;

  Error(Assignment, "Incompatible types in assignment") :-
      ~EquivTypes(Var.Type, Val.Type);
end Assignment;
```

Figure 4.23: Assignment statements.

the pragmas permitted within an operator. The `anchor` pragma indicates that the visit functions generated for an AST operator should be memoized, allowing a visit to be skipped during subsequent incremental re-analysis if it would necessarily result in an identical attribution. This optimization, called *visit caching*, allows unnecessary visits to be skipped but incurs a cost in both time and space. We thus place the resulting tradeoff under programmer control. The placement of the `anchor` pragmas controls the granularity of the incremental re-analysis.[1] We specify that declaration sequences should be re-analyzed at the granularity of a single declaration, visiting a declaration subsequence only when it contains one more more individual declarations requiring re-analysis.

4.10 Statements

The assignment statement, shown in Figure 4.23, is straightforward. The expression yielding the value to be assigned is checked for type compatibility with the variable receiving the value.

The block statement, shown in Figure 4.24, instantiates a `NormalContour` object to represent the binding contour introduced by the block. The surrounding binding context is the parent for the new block, and the bindings provided by the declaration list `Decls` are the bindings for the new contour. The body of the block receives the new contour object as its context.

A statement sequence, shown in Figure 4.25, distributes its binding environment to all of its constituent statements. The `anchor` pragmas indicate that the re-analysis of statements should take place at the granularity of an individual statement.

4.11 The Top-level Program

The root AST operator `Prog`, shown in Figure 4.26, establishes the global binding environment as the context for the statement sequence which comprises the top-level structure of a program in **Example**.

The analysis description for **Example** is now complete. For the convenience of the reader, it is reproduced without interspersed commentary in Appendix B.

[1]Unlike the other pragmas, which apply to attributes or attribute occurrences, `anchor` applies to an entire AST operator or child. We could have extended the pragma `maintained` to apply to simple attributes as well as functions and relations, but, since a visit cannot ever be skipped unless *all* of its inherited attributes are known to be unchanged (and would therefore have to be maintained between visits), the approach we have taken is less prone to user error.

```
operator Block : Statement is
  "declare"
      { Decls:Declarations }
  "begin"
      { Stmts:Statements }
  "end"
where
  Decls.Ctx = BodyCtx;

  object BodyCtx : NormalContour
  where
    Parent = Block.Ctx;
    Binds(Ident, Ent) :- Decls.Binds(Ident, Ent);
  end BodyCtx;

  Stmts.Ctx = BodyCtx;
end Block;
```

Figure 4.24: Block statements.

```
operator StmtList : Statements is
  { Stmts:Statement ";" }*
where
  Stmts.Ctx = StmtList.Ctx;

  analyze Stmts
  with
    context    Ctx : Environment;
  when [] =>
    % empty sequence
  when [ s ] =>
    % singleton sequence
    anchor;
    s.Ctx = Stmts.Ctx;
  when [ s1 ^ s2 ] =>
    % nondeterministic sequence split
    anchor;
    s1.Ctx = Stmts.Ctx;
    s2.Ctx = Stmts.Ctx;
  end;
end StmtList;
```

Figure 4.25: Statement sequences.

```
operator Prog : Program is
  { Body:Statements }
where
  Body.Ctx = GlobalEnv;
end Prog;
```

Figure 4.26: **Example** program top-level.

4.12 Discussion

Existing attribute grammar systems combine functional programming with attribution. The modern style of functional programming as embodied in Standard ML [54] and Haskell [40] makes extensive use of typed terms and pattern matching, and recent attribute grammar systems such as the Synthesizer Generator and FNC-2 [42] have followed suit. ADL is thus quite conventional in this respect, although we have borrowed a few notations from the logic programming tradition. Recently, Boyland [10] has designed and implemented an elaborate attribute grammar extension aimed at constructing complete language descriptions from the composition of smaller components. The underlying attribution paradigm is highly unconventional, and, while interesting on its own terms, poses new implementation challenges that we chose to avoid in our own work. ADL is thus thoroughly conventional in the monolithic structure of its descriptions. Tellingly, Boyland did not address either the generation of efficient (e.g., statically-scheduled) evaluators or incremental evaluation.

The inclusion of attributed object types with identity is a distinctive feature of ADL that provides an expressive mechanism for modeling language notions such as declared entities and binding contours. Attributed objects have appeared previously in Door Attribute Grammars [32, 33], but in a manner that exposed the imperative nature of their implementation. Our system is unique in providing objects in such a way that the user cannot compromise the integrity of the declarative semantics, understood as maintenance of constraint consistency. The need for object identity is clear in the case of declared entities. Analysis descriptions in existing attribution formalisms must either create explicit "unique identifiers" (UIDs) or press references to AST nodes into service toward the same end. Both of these solutions are inelegant. Creation of unique identifiers is difficult in a language without side-effects, and generally requires that a "next available UID" value be threaded throughout the tree. The use of node references as UIDs is a representational trick that conflates the essential properties of a language notion and an artifact of the syntax used to denote it. Within the implementation, of course, object addresses do serve as UIDs, but this is of no relevance to the user.

Relation-valued attributes are also a distinctive feature of ADL, and are useful in modeling many-to-many relationships where functions are not appropriate. In the example presented in this chapter, it was easy to accommodate the possibility that a given name might be multiply-declared within its contour. Without relations, we would have had to maintain an explicit list of bindings. Because the use of relations does not commit to the data structure used to represent them, the implementation is given more freedom to choose an appropriate one. Our treatment of relations as full-fledged attributes was anticipated by Sataluri and Fleck [69]. Horwitz [39] defines a notion of relational attribution that is akin to our collections. Her relations are global, however, and do not decorate the AST itself.

Throughout the discussion so far, we have hinted at the synergy between the ADL language design and the requirements of incremental execution. We devote the next four chapters to the methods employed in the analysis and translation of ADL.

Chapter 5

Incremental Visit-Sequence Evaluators

In the previous chapter, we showed how our extended attribution formalism, ADL, can be used to express the static semantic analysis of programming languages. We turn now to implementation issues, presenting the strategies we employ in the incremental execution of these specifications.

At the core of our implementation is an incremental evaluator for classical attribute grammars. Simple attributes in ADL, i.e., those introduced by the keywords `attribute` and `context`, are nothing more than those of classical attribute grammars. Furthermore, the implementations of the non-standard attributes, such as functions and relations, are most easily described as extensions to a traditional incremental evaluator. In this chapter, we present an incremental evaluation algorithm for classical attribute grammars that serves as the basis for the remainder of our work. This evaluator is in many ways unremarkable, being an adaptation of methods well known in the literature. Our evaluator is distinguished principally by its simplicity and straightforward design. Our design is based in part, however, upon the recognition and mitigation of a serious flaw in the usual representation of the abstract syntax tree in language-based environments, which has not, we believe, been given the attention that it deserves in the literature.

5.1 Visit-Sequence Evaluators

Attribute grammar systems intended for practical programming language analysis generally restrict the class of allowable attribute grammars in order to permit the use of efficient evaluation strategies. Our method is applicable to the ℓ-ordered [20] class of attribute grammars. These are defined as the attribute grammars for which there exists a family of total orders over the attributes of each phylum such that the attribute instances of each node in every AST can be evaluated in a sequence that respects the ordering for its phylum. As a consequence, the evaluation steps to be performed at each instance of a given AST operator can be fixed in advance, regardless of its context in the tree. In general, the remainder of the tree induces constraints on the evaluation order which cannot be determined by local examination of the node's attribute equations. In an ℓ-ordered AG, however, a conservative approximation to these constraints is always immediately apparent from the attribute ordering for the phylum to which the node belongs.

The ℓ-ordered class is more than adequate for practical purposes, as it contains all AGs evaluable in a fixed number of sweeps or passes as a proper subclass. Unfortunately, testing membership in the ℓ-ordered class been shown to be NP-complete by Engelfriet and Filè [20], and is regarded as impractical for AGs of realistic size. The difficulty arises in the computation of the required family of total orders on the attributes of each phylum. It is a relatively straightforward polynomial-time computation to determine a family of *partial* orders with the required property. It is an even simpler linear-time computation to test a given family of total orders for feasibility, and, if admissible, to generate the visit procedures. Unfortunately, however, there will in general be much freedom in converting the partial orders to the required total orders. Our implementation is thus based on the smaller, but polynomially-decidable class of ordered attribute grammars (OAGs), defined by Kastens [47].[1] The OAG class is defined by a greedy strategy for computing the total orders, placing each attribute as early as possible in the total order for its phylum, subject to the requirement that it follow all of the attributes that precede it in the corresponding partial order. (If more than one attribute could be chosen for a given position in the total order, then their relative order is arbitrary, and, as it turns out, without consequence to the outcome of the membership test.) The AG is then a member of the OAG class only in the case that the resulting family of total orders is feasible.

It is always possible to convert an ℓ-ordered AG to an OAG by adding additional attribute dependencies. In practice, only a few such dependencies, if any, are required. Thus the OAG class represents a pragmatic approximation to the ℓ-ordered class for the purpose of evaluator generation.

For any ℓ-ordered AG, it is possible to partition the attributes of each phylum X into a sequence of pairs of sets of inherited attributes and synthesized attributes

$$\langle I_1, S_1\rangle, \langle I_2, S_2\rangle, \ldots, \langle I_N, S_N\rangle,$$

such that all of the sets are non-empty except possibly for I_1 and S_N, and, in any tree, the instances of S_j depend only on those of $I_j \cup I_i \cup S_i, i < j$ and the instances of I_j depend only on those of $I_i \cup S_i, i < j$. In the case of OAGs, there exists a standard and efficient method to compute these sequences, due to Kastens [47]. The sequence of pairs, or *partition*, defines a sequence of visits to be made to each instance of phylum X during the traversal and attribution of the AST. Upon entry to visit j, the instances of the I_j will be available, as well as all instances computed in previous visits, thus the attributes S_j can be computed.

Given a partition for the phylum of an operator and those of its children, we can then construct a *visit sequence*, or *evaluation plan* for each operator. Following Reps and Teitelbaum [67], an evaluation plan is a sequence of instructions of one of the following kinds:

EVAL(i, a): Evaluate instance $i.a$, that is, the attribute a of child i. The attribute a must be a synthesized attribute if $i = 0$, that is, the attribute belongs to the AST operator itself (the LHS of the production), and it must be an inherited attribute if $i > 0$, that is, it belongs to a child (on the RHS of the production).

VISIT(i, r): Transfer control to the subtree at child i ($i > 0$) for visit number r.

SUSPEND(r): Return control to the parent node at the conclusion of visit r.

The last member of a visit sequence is always a SUSPEND instruction.

The evaluation plan for each operator forms a straight-line program. In the classical treatment, these programs are conceptually executed as coroutines, in which adjacent nodes receive and relinquish control via the VISIT and SUSPEND instructions. A naive realization based on a general

[1] There is some confusion in the terminology surrounding the ℓ-ordered AG and OAG classes. According to Engelfriet and Filé [20], Kastens originally used the term "ordered attribute grammar" in a 1978 technical report to refer to what we call the ℓ-ordered AGs, then later redefined the class in his 1980 paper [47] in order to give it a polynomial-time membership test. Engelfriet and Filé introduced the term ℓ-ordered, and we follow their lead. Their terminology was also adopted in the comprehensive survey of attribute grammars by Deransart *et al.* [17].

implementation of coroutines would be rather inefficient, however, as a separate program counter would be required for each node. In practice, the implementation is simplified by the fact that every VISIT and SUSPEND instruction contains a visit number. Instead of storing a separate program counter with each node, its value can be determined from the visit number upon each transfer of control, using an auxiliary table. A tree-walking evaluator of this kind requires no per-node overhead beyond the storage of the attribute values themselves, as the remainder of its state can be maintained in a few global variables.

In our evaluator, we use an alternative implementation strategy in which visits are mapped onto ordinary subroutine calls in the target language of our implementation [18, 49]. An evaluation plan can be decomposed into a series of segments, each terminated by a SUSPEND instruction, and representing the actions to be performed upon the i-th visit. We construct a set of *visit procedures* for each AST operator, one for each segment of the evaluation plan. For visit i, the visit procedure accepts the tree node and the attributes I_i as arguments, and returns the attributes S_i as results, invoking the visit procedures of the node's children as needed.

In general, the attribute values must be stored into the tree as they are computed, as subsequent visits may require access to those computed on previous visits. In practice, however, most attributes are *temporary*, meaning that an instance is used only within the dynamic extent of the visit procedure call that defines its value, and is not used within subsequent visits to the same node. Experience with implemented attribute grammar systems has shown that more than 90 percent of the attributes in a typical attribute grammar are temporary, according to Julié and Parigot [44], who report a series of experiments with the FNC-2 system in which only 7.5 percent of the attributes (weighted by their number of occurrences) required storage in the tree. The instances of temporary attributes can be allocated space on an auxiliary stack during the tree traversal, greatly reducing the size of the AST and the overall storage requirement. In our evaluator, the runtime stack maintained by our target language serves as the temporary stack, giving us this optimization almost without effort.

5.2 Incremental Evaluation

When the AST is modified, we wish to restore attribute consistency as quickly as possible, exploiting the fact that many attribute instances may retain their old values. In the simplest case, a tree modification consists of the replacement of a single subtree with another. Because a special "placeholder" node always appears wherever a subtree is missing, the subtree replacement operation subsumes both elaboration of a previously unexpanded subtree and subtree deletion. An attribute x_0 defined by the equation $x_0 = f(x_1, x_2, \ldots, x_N)$ is said to be *inconsistent* if its current value is not that of the function f applied to the current values of $x_1, x_2, \ldots, x_N$. A subtree replacement potentially introduces one or more inconsistent attribute instances at the interface between the new subtree and its parent node. In a classical incremental evaluator, changes are propagated outward in the attribute dependency graph from the initial point of inconsistency. Re-evaluating an attribute may cause one or more of its immediate successors to become inconsistent as well. Propagation stops when no more inconsistent attributes remain.

Under the assumption of a single-subtree replacement, is possible to guarantee that the number of attribute re-evaluations, as well as the associated bookkeeping overhead, is bounded by $O(|\mathrm{AFFECTED}|)$, where AFFECTED is the set of attribute instances whose values differ before and after re-evaluation. This performance criterion is nearly universally cited as the goal to which an incremental attribute evaluator should aspire, and is termed *(asymptotic) optimal-time* by Reps, Teitelbaum, and Demers [66]. Algorithms that achieve this complexity generally do so by assuring that an attribute is evaluated only after its predecessors have achieved their correct final values, i.e., change propagation respects a topological ordering with respect to the attribute dependency graph. Failure to observe this requirement may lead to unnecessary and redundant re-evaluation of the attribute instances corresponding to large regions of the dependency graph. In fact, Reps *et al.* show that naive change propagation can exhibit exponential-time behavior for some attribute grammars.

In the same paper, Reps *et al.* describe an incremental evaluator that works for any noncircular AG, and runs in asymptotically optimal time. Scheduling is performed entirely at runtime, without any prior analysis of the attribute dependencies, thus resulting in poor performance. The use of explicit attribute dependency graphs at runtime results in high space consumption as well. In later work [67], Reps and Teitelbaum develop another algorithm, applicable only to ordered attribute grammars, as a variant of the classical non-incremental tree-walking (coroutining) evaluator. The essential idea is that a visit to a node may be skipped if it is known that the node contains no inconsistent attributes. Since a visit to a node always re-evaluates all attributes scheduled at evaluator generation time for that visit, the new algorithm may evaluate more attributes than the older one. Nonetheless, both the number of attribute re-evaluations and the bookkeeping overhead remain $O(|\mathrm{AFFECTED}|)$. The topological ordering required to assure optimal-time evaluation is implicit in the statically-computed evaluation schedule. Since dependency graphs need not be maintained at runtime, overhead is greatly reduced.

Vogt, Swierstra, and Kuiper [75] have investigated incremental attribution of OAGs in a functional setting. In this case, an AST cannot be modified, but new trees may be created that share most of their structure with an existing one. The role of incrementality is then to attribute the new tree efficiently by reusing the attribution of others with which it shares structure. Since a tree itself may not be modified, attributes must be stored externally in an associative store. These considerations lead naturally to the idea of caching or "memoizing" the visit functions. Subsequent visits to a given subtree with the same inherited attribute values as on the previous visit simply return the previously computed synthesized attributes. Non-temporary attributes are awkward to handle in a functional context, requiring that they be collected in a temporary tree structure called a *binding* which is then passed as an argument to subsequent visits as if it were an additional inherited attribute. (See Vogt *et al.* [75] for details.) Performing a subtree replacement requires that all nodes on the path from the edit site to the root be created afresh, precluding optimal-time performance.

5.3 Multiple Subtree Replacements

In practice, the restriction to a single subtree replacement is inadequate. Language-based environments incorporating incremental parsing (as opposed to direct editing of tree structure) may generate many subtree replacements from a single textual change. Furthermore, a set of related changes may be viewed as a unit by the user, who may wish to delay re-analysis until that unit is complete. The asymptotically optimal methods previously discussed, when applied after multiple subtree replacements have taken place, will propagate changes from each replacement site independently. Should the affected regions of the tree intersect, the topological ordering constraint may be violated, and the affected regions may have to be re-evaluated. Reps *et al.* [66] show that naive change propagation can be exponential in $|\mathrm{AFFECTED}|$ for a depth-first traversal of the dependency graph, and quadratic if a breadth-first traversal is used. Good performance in the general case requires that change propagation originating from multiple initial sites of attribute inconsistency be *coordinated* in order to avoid this behavior.

Several authors have addressed this issue, most notably Peckham [57, 58], who developed an algorithm that maintains a favorable amortized complexity of $O(k \log n + |\mathrm{AFFECTED}|)$ in the presence of multiple subtree replacements, where k is the number of subtree replacements and n is the total size of the tree. Actually, Peckham only proves an amortized complexity of $O(\log n \cdot |\text{AFFECTED}|)$, and the bound cited previously is given only as a plausibly-argued conjecture. Unfortunately, the method incurs a large amount of administrative overhead in the form of auxiliary data structures in comparison to a comparable evaluator for the restricted editing model. Furthermore, Peckham's method is applicable only to a subclass of the ordered attribute grammars whose size and character has not been adequately assessed.

An alternative approach, suggested by Reps [65], merges the separate regions of attribute inconsistency into a single connected one. Before beginning re-evaluation, all previously computed attributes along paths connecting the edit sites are invalidated, resetting them to their original uninitialized state. In practice, a single bit within each node represents the validity status of its attributes. While it suffices to invalidate the attributes on a path from each edit site to their mutual least common ancestor, it is often more convenient simply to clear a path all the way to the root. Using this method of coordination, the complexity of the incremental evaluator becomes $O(|\text{AFFECTED}| + |\text{EDIT\_ANCESTORS}|)$, where $|\text{EDIT\_ANCESTORS}|$ is the number of nodes on all paths from edit sites to the root of the tree.[2] Peckham notes that the coordination overhead in this approach is bounded only by the size of the tree, as abstract syntax trees (ASTs) in language-based environments typically are unbalanced ([58], page 2).

The functional evaluator of Vogt *et al.* is inherently coordinated, as it implicitly embodies Reps' simple coordination strategy. The allocation of fresh tree nodes along the spine above each subtree replacement is analogous to attribute invalidation. It is therefore not surprising that the method achieves the same time complexity, $O(|\text{AFFECTED}| + |\text{EDIT\_ANCESTORS}|)$ [75].

5.4 Our Incremental Evaluator

Our evaluator is based on visit procedures. To make the evaluator incremental, we cache ("memoize") the calls to the visit procedures in a manner reminiscent of the functional evaluator of Vogt *et al.* In a purely functional setting, the result of a function depends only on the values of its arguments, so if a function has been called with the same arguments as before, it can simply return the same result with no further effort. The caching of visit procedures is complicated, however, by imperative modifications to the tree argument due to editing and storage of attribute values. The overhead of a structural equality test on trees would be prohibitive, so we must provide an alternate means to account for the dependency of the visit procedure results on the tree argument. We provide each tree node with a *subtree-modified* bit, which, at the beginning of the analysis, is set in every node such that the subtree issuing from that node has been modified. Whenever the editor modifies the value of a child pointer in the tree, it sets the subtree-modified bit in the node containing that pointer and in all of its ancestors. If the traversal reaches a node with the subtree-modified bit set, the visit must be performed regardless of the inherited attribute values. If the subtree-modified bit is not set, however, the incoming inherited attribute values are compared with those saved from the previous update, and, if equal, the synthesized attributes computed previously are returned. If the arguments are not equal, then the visit takes place. Since visits may refer back to attributes computed in previous visits and stored in the tree, once a visit has been made to a node (reflecting a possible change to an attribute value stored in the node), the subtree-modified bit is set to force all subsequent visits. The bit is cleared upon exit from the last visit to each node.

A code template for incremental visit procedures is shown in Figure 5.1. The procedure used for marking changes to edited nodes is shown in Figure 5.2. These examples, and further examples

[2] Vogt *et al.* use the misleading term PATHS_TO_ROOT.

of the generated target code appearing in this dissertation, are rendered in an Algol-like syntax that is intended to be self-explanatory. The language is simply a rendering of Lisp with a more conventional syntax and a simplified object system, and is dynamically-typed. Functions introduced with the keyword **method** are associated with a class of objects, and execute with the pseudovariable **self** bound to the instance of the class relative to which the method was called. The class to which a method belongs should be clear from the explanatory text, and is often an AST operator.

The time complexity of our evaluator is $O(|\text{AFFECTED}| + |\text{EDIT\_ANCESTORS}|)$. However, our evaluator is able to handle multiple edit sites without additional cost, while all of the published optimal algorithms assume a single subtree replacement. Modifying any of the optimal methods to use the simple coordination method suggested by Reps yields the same complexity as our algorithm.

The algorithm as it stands does not exhibit robustly scalable incremental performance, as $|\text{EDIT\_ANCESTORS}|$ is bounded only by the size of the AST. In practice, deeply-nested structures arise in ASTs as an artifact of the recursive representation of sequences, usually as linear lists. For other constructs, which appear to the user in nested form, there is a practical bound on the depth of nesting, imposed by cognitive and stylistic concerns. By representing sequences as balanced binary trees, we keep the AST approximately balanced, such that $|\text{EDIT\_ANCESTORS}|$ is bounded by $O(k \log N)$, where N is the total size of the tree and k is the number of edit sites. This yields a practical time complexity of $O(k \log N + |\text{AFFECTED}|)$.

5.5 Balancing the Abstract Syntax Tree

While any recursively-defined phylum may generate a tree of unbounded size in theory, the depth of recursive structures is bounded in practice by the user's ability to keep track of the nesting of language constructs. The exceptions to this principle are constructs that are conceived of by the user as repetitive in structure, i.e., as sequences, even though their internal representation may be recursive.[3] It is conventional to represent unbounded sequences in an AST as linearly recursive lists. As a consequence, the AST can be arbitrarily unbalanced. We propose an alternative representation of sequences as balanced binary trees, in which the leaves of the tree represent the elements of the sequence. Under the practical assumptions of the previous paragraph, this is sufficient to guarantee that the depth of the tree is bounded by $O(\log n)$, in which case the simple spine-breaking method of coordination achieves the same asymptotic complexity as Peckham's complex algorithm.

5.5.1 Balanced Sequences and |AFFECTED|

Indeed, balancing the AST mitigates a more general problem with the attribution of sequences that reveals a serious limitation of the $O(|AFFECTED|)$ optimality criterion. Peckham's bound, when applied to the attribution of an unbalanced sequence, may conceal in its $O(|\text{AFFECTED}|)$ term a comparable performance penalty to that incurred by our algorithm on the same tree. Making the reasonable assumption that at least one synthesized attribute of the sequence depends on an attribute of each of its children, any attribute grammar performing this attribution on a right- (left-) recursive linear list will necessarily include in every internal node of the sequence an attribute whose value is dependent on an attribute of the last (first) element of the sequence. Thus the value of $|\text{AFFECTED}|$ is $O(n)$ for such an attribution. Now suppose that the attribution in question merely

[3] Language constructs such as `if–then–else` are sometimes used in a deeply-nested but idiomatic manner reminiscent of sequences, e.g., in Pascal to simulate a `case` statement for a non-scalar type. We overlooked this because our work focused on Modula-2, which permits an `elsif` clause. In languages where these idioms exist, the parser must recognize them and convert them to a form that makes their role as sequences apparent, using special AST operators provided for this purpose. We thank Mark Wegman for this observation.

```
% A typical visit procedure, for each visit but the last.

function SomeOperator.VISIT_i(a₁, a₂, ..., aₙ)
  if not self.SUBTREE_MODIFIED
    and self.I₁ == a₁
    ...
    and self.Iₙ == aₙ
  then  % Visit cache hit.
    % Return synthesized attributes computed previously.
    return (self.S₁, self.S₂, ..., self.Sₘ)
  else  % Visit cache miss.
    % Force subsequent visits, even if inherited attributes are the same.
    self.SUBTREE_MODIFIED = true
    % Store inherited attributes.
    self.I₁ = a₁
    ...
    self.Iₙ = aₙ
    % Compute and store synthesized attributes.
    self.S₁ = ...
    ...
    self.Sₘ = ...
    % Return synthesized attributes just computed.
    return (self.S₁, self.S₂, ..., self.Sₘ)


% The last visit is handled similarly, except for SUBTREE_MODIFIED.

function SomeOperator.VISIT_N(a₁, a₂, ..., aₙ)
  if not self.SUBTREE_MODIFIED
    and self.I₁ == a₁
    ...
    and self.Iₙ == aₙ
  then
    return (self.S₁, self.S₂, ..., self.Sₘ)
  else
    self.I₁ = a₁
    ...
    self.Iₙ = aₙ
    self.S₁ = ...
    ...
    self.Sₘ = ...
    % All attributes in this node are now up-to-date.
    self.SUBTREE_MODIFIED = false
    return (self.S₁, self.S₂, ..., self.Sₘ)
```

Figure 5.1: Incremental visit procedures.

```
function Notify_Change(node)
  unless node = nil then
    node.SUBTREE_MODIFIED = true
    Notify_Change(node.parent)
```

Figure 5.2: Marking a node and its ancestors when modified.

constructed a list of the child attribute values. Then the same attribution could be computed on a balanced tree such that |AFFECTED| is $O(\log n)$, using list concatenation (`append`) instead of addition of a single element (`cons`). The difference is entirely an artifact of the particular attribute grammar used to express the computation, not any inherent property of the problem to be solved, and is forced upon us solely by an inappropriate choice of representation for sequences.

We must be careful in making this sort of argument, as merely counting attribute evaluations may be misleading when the attribute domains include values of unbounded size. If we were to represent the list attribute value as a linear list, we would simply move execution time from attribute evaluation overhead into the concatenation operations. To realize a genuine improvement, we may use a list representation in which the link cells represent the `append` operation, rather than `cons`, yielding a constant-time implementation of `append`. In general, a reduction in the size of AFFECTED due to balancing will be observed whenever synthesized attributes of sequence children are collected and combined via an associative combining operator, but an actual improvement in runtime may depend on appropriate data structure and algorithm choices.

5.5.2 Representing Balanced Sequences

Each occurrence of a sequence must belong to a sequence phylum, which represents a sequence of subtrees of identical phylum that are to be attributed in a particular way. For each sequence phylum, we construct three operators that are used to represent its instances:

- The *null* operator represents an empty sequence, and has no children. It serves also as the completing operator for the sequence phylum, used as a placeholder when a parsing error prevents constructing an AST for the sequence.

- The *singleton* operator represents a sequence containing a single element. It has one child, which belongs to the element phylum of the sequence.

- The *pair* operator represents the concatenation of two non-empty subsequences. It has two children, either or both of which may be singleton or pair nodes. The null operator may not appear as a child of the pair operator.

The attribute grammar formalism forces us to choose a structurally-recursive representation for the abstract notion of a sequence, as attribution is defined to take place on trees consisting of nodes of fixed arity. This choice of representation will largely determine the attribute equations required in order to perform a given computation, in particular, the manner in which information is routed through the tree. It can be shown that any attribution applicable to a linearly-recursive list can be rendered in our doubly-recursive representation as well. In the linear list representation, attributes may be directly transmitted only between adjacent siblings in the sequence, though this communication is realized via parent-child attribute flow. In our doubly-recursive representation, by threading each such attribute through the tree via copy rules in the manner of a right-to-left or left-to-right traversal, attributes may be passed between conceptually adjacent siblings regardless of the actual shape of the tree. It is clear, however, that using this method to mechanically translate attributions designed with a linear-list representation in mind will gain nothing in performance, as spurious sibling-sibling dependencies in the original formulation will simply be rendered as similar dependencies in the translated version. In fact, we will pay an extra price for the copy rules.

It is advisable, then, to consider the problem of sequence attribution in a manner uncommitted to a particular representation, and then to ask how efficiently various patterns of information flow can be rendered using the sequence representations under consideration. Nord and Pfenning [55] identify four important patterns of attribute flow in sequences:[4]

1. Construction of a list containing the value of a given synthesized attribute at each child.

2. Distribution of an inherited attribute of the sequence to all children.

3. Chains of inherited/synthesized attributes threaded through identically-named instances left-to right.

4. Chains of inherited/synthesized attributes threaded through identically-named instances right-to-left.

The first pattern of attribution always benefits from the balanced representation in terms of the size of AFFECTED. In this case, the number of attributes that must be re-evaluated in order to propagate an attribute value change at a sequence element to the root of the sequence is $O(\log n)$, as compared to $O(n)$ for a linear list, where n is the length of the sequence. This property is due entirely to the associativity of the list concatenation operator, and extends to reduction of the sequence of element attribute values by other associative operators such as union, intersection, addition, and multiplication.

For the second pattern of attribution, it is impossible to do better than $O(n)$, as the new value must be transmitted to each of the n sequence elements. In our representation, there is a constant-factor increase in the number of copy attributes due to the use of an explicit singleton node for each sequence element. It would be possible to eliminate the singleton nodes, folding them into their parent pair nodes, but this would require distinguishing four distinct varieties of pair nodes in order to properly allocate space therein for the attributes of the elided singleton nodes.

The third and fourth patterns of attribution involve sibling-sibling communication not arising as an artifact of the tree representation. Again, our representation suffers a constant factor penalty, and yields the same asymptotic behavior. It is worth noting that sibling-sibling communication patterns necessarily result in |AFFECTED| being $O(n)$, and thus should be avoided whenever possible in an incremental application.

Since any attribution of a linear list will require sibling-sibling communication, hence left-to-right or right-to-left threading, our approach is at worst equivalent from an asymptotic point of view, and results in a drastic improvement for one important case. There is a constant-factor penalty due the the presence of singleton nodes, but even this may be eliminated at the cost of a small increase in implementation complexity.

[4] These are taken from a set of six such patterns identified by Jullig and DeRemer [45] in connection with a notation for attributing regular right-part grammars. The two patterns not shown here reflect constraints as well as attribute propagation, as Jullig and DeRemer were working in the framework of Extended Attribute Grammars [81], a restricted variant of two-level grammars.

constraint $\rightarrow$

```
analyze child_name
with
  { attribute_definition }+
when [ ] =>
  { constraint }+
when [ singleton_name ] =>
  { constraint }+
when [ left_subseq_name ^ right_subseq_name ] =>
  { constraint }+
end ;
```

Figure 5.3: The `analyze` construct.

We use AVL trees for balancing. While it is possible to implement AVL trees using a three-state flag to record balance information, we maintain the actual depth of each subtree. This facilitates an efficient $O(\log n)$ algorithm for sequence concatenation, which is required by the parser.[5] We also keep track of the number of leaves below each node, which allows $O(\log n)$ access to any child given its index in the sequence. This operation is not used during attribution, but is required by user-level tree navigation commands.

5.5.3 A Notation for Attributing Balanced Sequences

We have designed a specific notation that encourages a somewhat more abstract view of sequences, but in fact maps directly onto our concrete sequence representation. In our language, sequence phyla are declared implicitly. Any child of an operator may be marked as a sequence, in which case a sequence phylum and its operators are created. The `analyze` construct, introduced in the previous chapter, occurs syntactically in the role of an attribute equation, and defines a "local" attribution of the anonymous sequence phylum and its operators. Its precise syntax is given in Figure 5.3.

Within the `analyze` construct, the *child_name* refers to the sequence as a whole, and has attributes as defined by the *attribute_definition* sequence. There are three branches, representing a case analysis. If the sequence is empty, the first set of constraints applies. If the sequence is a singleton, then the element is known by the name *singleton_name*, and the second set of constraints applies. Otherwise, the sequence consists of two or more elements, and can be viewed as the concatenation of two non-empty sequences. The third case then represents a non-deterministic split of the sequence into two subsequences bound to the names *left_subseq_name* and *right_subseq_name*. Figure 4.22, in the previous chapter, illustrates the usage of the `analyze` construct. There, the context reaching a sequence of declarations is passed downward to each of its constituent declarations, and the bindings are passed upward and accumulated.

[5] During modification of the existing incremental parser, we noted that it could exhibit degenerate linear-time behavior (in the total size of a sequence, hence the entire program) while parsing a change to a sequence element. This problem is inherent in most incremental parsing algorithms when linear lists are used for sequences. These parsers "unzip" the tree along the spine from each edit site, and then attempt to reassemble the fragments. For a highly-unbalanced tree, the number of fragments may be on the order of the length of the program, even for a small edit. Adopting the balanced tree representation corrected this problem, which arose in precisely the same way as the problem with attribution.

Our notation favors the use of parent-child rather than sibling-sibling patterns of communication by allowing a more compact form of expression in the former case. This is in contrast to the linear list representation, which makes all patterns of attribution equally convenient (or inconvenient, as the case may be). Furthermore, the semantics of the nondeterministic split forces the use of an associative combining operator. Of course, it is possible to evade the spirit of the `analyze` construct with sufficiently complex attributes and semantic functions, but additional effort will be required to do so.

5.6 Selective Visit Caching

Most published incremental evaluation algorithms have implicitly assumed that all attribute instances are stored in the tree, and have fixed a fine level of granularity for attribute re-evaluation. Storing all of the attributes is costly in terms of space, and limiting space consumption by the attributed AST has been a widely recognized but unsatisfactorily addressed engineering concern in existing programming environments based on attribute grammars. In contrast, hand-coded incremental language-based environments have favored re-analysis at the level of entire source-language procedures. Feiler [24] and Ross [68] describe systems of this kind.

In a non-incremental attribute evaluator, many attributes are of use only transiently during attribute evaluation, and need not be present in the final attributed tree. It is usually possible to allocate these attributes in one or more auxiliary stacks or global variables used only while evaluation is in progress. Farrow and Yellin [21] compare several methods for storage optimizations of this kind, and Julié and Parigot [44] present more recent results as employed in the FNC-2 system. In some cases, such as for the one-sweep [19] and ℓ-attributed classes of attribute grammars, *all* attributes may be allocated in this way, provided that we are interested only in the values of one or more synthesized attributes of the root. An incremental evaluator must retain *some* attribute values between re-evaluations in order to permit the reuse of previous analysis. It does not follow, however, that all attributes should be maintained. While it clearly makes sense to maintain the results of expensive calculations, many attributes are cheaply recomputed.

In our evaluator, otherwise temporary attributes are retained by the visit-caching mechanism. Visit caching comes at a cost in both time and space, and will not always pay off in improved performance. The granularity with which re-evaluation occurs, i.e., the extent to which extraneous attributes are allowed to be re-evaluated before quiescence is detected, determines an important time-space tradeoff which should be under the control of the author of the analysis description. The time and space overheads of avoiding re-evaluation may outweigh the cost of doing the evaluation for simple computations over subtrees of (practically) bounded size. For example, we might set the granularity at the level of an entire expression (i.e., the subtree below the point at which the expression adjoins a larger non-expression construct), and forgo the possibility of re-evaluating attributes for only a part of the expression. Our implementation allows visit caching to be enabled selectively for each operator or for particular child positions within an operator. In the latter case, the cache test is performed at the call site rather than upon entry to the visit procedure of the child. In this way, the granularity of re-analysis can be contextually determined.

5.7 Summary and Related Work

Incremental attribution for static analysis of programming languages is an established research paradigm. Our work exploits many ideas that have appeared in one form or another in the previous research literature. In comparison, however, our approach is in each case either simpler or more general.

In their seminal paper, Reps *et al.* [66] defined the $O(|\text{AFFECTED}|)$ optimality criterion and the strategy of change propagation in topological order that achieves it. Their algorithm, employed in the Synthesizer Generator [67], works for arbitrary noncircular attribute grammars, but the overhead in space and time is prohibitive. In later work, a simpler asymptotically-optimal evaluator for OAGs was developed, and its use recommended. Our evaluator was obtained by adapting to an imperative setting an alternative scheme for OAGs based on visit caching developed by Vogt *et al.* [75]. The resulting algorithm is quite similar in effect to the OAG evaluator used in the Synthesizer Generator, but the visit caching viewpoint leads to a more straightforward implementation.

The coordinated evaluators described by Reps, Marceau, and Teitelbaum [64], and Peckham [57] use additional static analysis, AG class restrictions, and complex auxiliary runtime data structures to account for dependencies linking disconnected inconsistent regions of the AST, thus assuring that evaluation always proceeds in topological order. We adopt the much simpler solution of merging the inconsistent regions at $O(|\text{EDIT\_ANCESTORS}|)$ cost, a natural consequence of the visit-caching approach.

Under practical assumptions about the depth of ASTs, using a balanced representation for sequences, the asymptotic complexity is as least as good as any other published algorithm. The idea of keeping an AST approximately balanced using a balanced sequence representation was suggested by Pugh [61] in connection with a function-caching incremental AG evaluator similar to that of Vogt, *et al.* Gafter [28] independently proposed the same technique in connection with parallel execution of a compiler, in which balanced sequences lead to a better decomposition of the program into subproblems. Gafter explicitly addressed incremental parsing with balanced sequences. Surprisingly, no one seems to have previously observed the implications of this technique for efficient coordinated evaluation, despite the fact that both Reps *et al.* [64] and Peckham [57] use other balanced data structures.

User-supplied pragmatic annotations to control re-evaluation granularity were previously implemented in OPTRAN [50], which used an incremental evaluation strategy similar to our own.

Although we have presented our algorithm as an evaluator for the ℓ-ordered class of attribute grammars, it is in fact applicable to any class of AGs for which the required visit procedures can be constructed, including the strongly-noncircular (SNC) attribute grammars. This class is of great importance because it appears to include all noncircular attribute grammars that arise in practice, and admits a polynomial-time membership test (as do OAGs). For an SNC AG, the order in which the attributes of a node must be evaluated depends in general on its superior context in the AST, thus multiple visit sequences must be generated for each AST operator, unlike the case of an ℓ-ordered AG in which a single visit sequence suffices. Jourdan and Parigot [43] outline a practical method for constructing visit procedures from an SNC attribute grammar, as employed in the implemented FNC-2 system.

In the next chapter, we extend our classical attribute evaluator to propagate non-local attribute dependencies efficiently.

Chapter 6

Objects and References

Objects are used to model programming language notions such as declared entities and scopes, where it is meaningful to distinguish different occurrences of entities that are otherwise functionally equivalent. Objects are accessed via object references, a special kind of value that refers indirectly to an object. Properties of the entities represented by an object are modeled by its *components*, a set of named attributes that are accessible via selection wherever a reference to the object is known. Every object belongs to a class, which defines the name and type of each of its components. Classes are arranged in a hierarchy modeling a taxonomical classification of the entities that the objects of the class represent.

In this chapter, we will treat objects as simple records without functional and relational components – i.e., without "methods," which will be treated in the following two chapters.

6.1 Objects and Non-local Dependencies

The values of the components of an object may change while preserving the identity of the object to which they belong. Our implementation exploits this semantics in order to propagate changes to object components efficiently. Object references, transmitted as ordinary attribute values, establish a link between the site at which the object is created and the sites at which the components of the object are selected and examined. Selections establish additional hidden bookkeeping links in the reverse direction, pointing from the selected components to the sites at which the selections occur. Changes to component values are immediately available at the selection sites, which are then "notified" via the hidden links, forcing the re-evaluation of any attributes dependent upon the result of the selections. Selections involving a component that is not changed need not be notified. If the identity of the object from which a selection takes place, i.e., the value of the object reference, changes, the selection is also re-evaluated, as the selection depends on the object reference in the conventional way.

In a conventional attribute grammar, the direct successors of a given attribute instance must be within the same AST node or within an adjacent node. In contrast, a component selection may be located at an arbitrary distance within the AST from the object instantiation in which the selected component is defined. The dependency of a selection upon its selected component may thus span an arbitrary distance within the tree. The *non-local* attribute dependencies maintained in the implementation of object instantiation and component selection thus provide an efficient realization of long-distance attribute dependencies, such as those that arise between the declaration of a named entity and its uses within the program. In the analyzer for **Example** developed in Chapter 4, variable declarations (Figure 4.20) result in the instantiation of an object:

```
object VarObj : VarEntity
where
  Type = Ty.Type;
  ...
end VarObj;
```

In a variable usage (Figure 4.14), a reference to the object representing the variable is extracted from the binding environment:

```
attribute Ent : BindingStatus =
    SimpleVar.Ctx.VisibleBinding(Name.Text);
```

The type of the variable is then obtained from the reference by selection:

```
SimpleVar.Type = Ent.Type :-
    Ent isa VarEntity;
...
```

A change to the type of the variable as it appears within the variable declaration will be propagated directly to the variable references at which the variable is accessed. The binding contour will not be affected, as the *identity* of the variable remains unchanged.

Objects also provide a way to bypass some copy rules, allowing the node-by-node propagation of a single object reference to stand in for the similar propagation of each component individually. The representation of binding contours in **Example** used objects in this way, allowing the single inherited attribute `Ctx` to stand in for the separate propagation of the `LocalBinding`, `VisibleBinding`, and `Duplicate` attributes to each place where a name might need to be declared or resolved. In fact, since inherited functional attributes are not permitted, the use of a contour object would be forced upon us even if it were a circumlocution.

6.2 Static Allocation of Objects

An object is created by an object instantiation, a special kind of attribute constraint that binds a name to a reference to a new object of the given class. Since there is no other way to create an object, objects cannot be created dynamically, e.g., during the evaluation of another attribute. Each object instantiation appearing in the constraints of an AST operator induces exactly one instance of the object in each node in the tree labeled by that operator. This enables us to allocate storage for objects statically within the containing tree node. From the viewpoint of a remote component access via a reference to the object, the object is a self-sufficient entity that can be interpreted knowing only its class. From the point of view of the node that contains it, components of the object are just ordinary local attributes.

Objects in which all components are simple attributes, i.e., are neither functions nor relations, can be used to model tuples. It might appear that tuples are redundant, then, and could be eliminated. In fact, the essential semantics of objects and tuples are quite different, as is their pragmatic treatment in our implementation. Two tuples are considered equal when their respective components are equal, whereas objects each have an identity distinct from any other structurally-isomorphic value. Conceptually, tuples follow "value semantics," i.e., the value of a tuple depends on those of all of its components, in contrast to object references, which follow "reference semantics," i.e., the value of the reference depends only on the identity of the object to which it refers. In principle, it would be possible to implement tuples in such a way that changes to a component of a tuple were propagated directly to the places where that component is actually observed, but in practice, the generality of the tuple constructor, which may appear in any expression, makes this very difficult. In contrast, the static allocation of objects allows us to implement a relatively simple strategy for maintaining the connection between an object component and the selection sites that access precisely that component. Restricting the fine-grained tracking of component dependencies to object types has not been a problem for us in practice, and agrees in an aesthetically pleasing way with the differences in the conceptual nature of the component dependencies.

6.3 Maintaining Dynamic Dependency Traces

Since the object reference from which a given component is selected may be the outcome of an arbitrary computation, we cannot predict in general which selection sites will depend on which components. These dependencies must be represented at runtime by dynamically maintained data structures. The dynamic dependencies will always be covered, however, by a simple approximation in which the object reference bound by an object instantiation is assumed to depend on the value of every component. We use this worst-case static approximation in order to permit a generation-time circularity test, and to schedule visits to the nodes containing the object instantiations and component selections in such a way that the worst-case dependencies that might arise at runtime are respected.

Each component of an object has an associated *non-local dependency list* (NLDL), whose members refer to the selection sites that currently access the component. When a component value changes, the nodes containing selections that access that attribute are *notified*. A notified node is marked for future visits, including the marking of its ancestors, in exactly the same manner as if a child of the node had been replaced. For this reason, it would be necessary that our basic incremental evaluation method support multiple initial sites of attribute inconsistency even if our editing model did not include multiple subtree replacements. Because the visit procedures are generated based on the static dependencies, we are assured that the evaluation traversal has not yet reached the remote component selection, even though it may already be under way at the time the node containing the selection is notified. Either the node is being re-evaluated already (at least one visit has entered it), or the subtree-modified bit will have been set before the traversal arrives at the node for the first time.

Re-evaluation may result in the removal of old dependencies as well as the addition of new ones. Node deletion may also require that notifications or updates to the contents of one or more NLDLs be made. Our strategy for performing the necessary bookkeeping distinguishes two kinds of selections, based on their syntactic form: selections from an attribute name, and selection from a bound variable. Since each attribute takes on a single value, each textual occurrence of a component selection from such an attribute must necessarily refer to a single component instance. For these *static* selections, we can determine the number of dynamic dependency links that will be needed by a trivial examination of the analysis description source. In contrast, a single textual occurrence of a variable may take on multiple values during a single node visit, e.g., an argument to a recursive function. A component selection from such a variable may thus give rise to an arbitrary number of dynamic dependency links as the variable takes on successive reference values. These selections are referred to as *dynamic*, and are handled in a different manner. Often, a variable can be shown to take on a single value, in which case a selection from the variable may be profitably treated as static. The current classification based on a trivial syntactic property is admittedly crude.

6.4 Static Component Selections

Each textual occurrence of a static selection is assigned a numeric index, relative to the containing AST operator, at evaluator-generation time. A non-local dependency list (NLDL) is allocated for each object component within the AST node that contains the object. The elements of the NLDL indicate each dependent selection site as a $\langle$*node, index*$\rangle$ pair. The NLDL is doubly-linked, so that any element can be removed in constant time. The insertion and deletion operations are simplified by linking the list in a ring, using a dummy header element which is allocated at the time the AST node is created. Each AST node that contains selections has a *selection registration vector*, with one entry for each selection. Each entry is a pointer back to the NLDL element that refers to the selection. These data structures are illustrated in Figure 6.1.

In addition to the NLDL, each component has an associated *registration method*, implicitly declared in the class and instantiated in each object by overriding a dummy method inherited from the class. The registration method takes a node and a selection index as an argument, and adds an entry to the NLDL of the component. Upon the evaluation of a reference-valued attribute, all selections using that attribute as the base are registered by calling the registration method associated with the selected component. The node that is currently being visited, i.e., the one in which the selection takes place, is passed as an argument, along with a statically-assigned selection index. Figure 6.2 shows how the selection of the `Type` component is performed within `SimpleVar`.

When invoked, the registration method allocates a new element in the NLDL of the selected component, pointing to the current selection, and updates the selection registration vector entry to point to the new element. It then uses the backpointer stored in the selection registration vector to find the NLDL element that currently points to the selection and remove it from its (unknown) NLDL. Figure 6.3 shows the definition of the registration method for the `VarObj.Type` component.

The protocol is robust in the face of node creation and deletion, and does not leave dangling links. When a node is deleted, which occurs prior to the beginning of the traversal, the NLDL elements that point to it are removed via the selection registration vector. Likewise, any nodes that contain references to an object in a deleted node will be visited during evaluation, as the referent object must necessarily have changed, and the normal registration process will then update the backpointers in the selection registration vectors. (It is possible that a deleted node will be notified. This does no harm, as the link to the parent of the deleted node will have been reset to **nil**, preventing the erroneous invalidation of any ancestors still in the tree.) Figure 6.4 shows how the node destructor for `SimpleVar` nodes handles its embedded NLDL.

As we shall see in the sequel, it is sometimes useful to know at the site of a selection from an attribute whether the component value actually changed, even though our general strategy is to retry all computations at the granularity of an entire visit. To avoid having to maintain a copy of the selected value in the node that performs the selection, we arrange for the registration method to return a boolean value that indicates whether the component has changed since the previous time the registration was performed. This can be done without the allocation of additional storage by shifting the responsibility for deleting the old NLDL entry onto the notification procedure, and storing a null value in the selection registration vector during notification in order to identify the particular selection that has been invalidated by the change. We say that such a selection has been *notified*. Notification of a selection always implies notification of the node in which it is contained. By verifying that the base object reference is unchanged and that the selection has not been notified, we can be assured that the value of a selection remains unchanged.

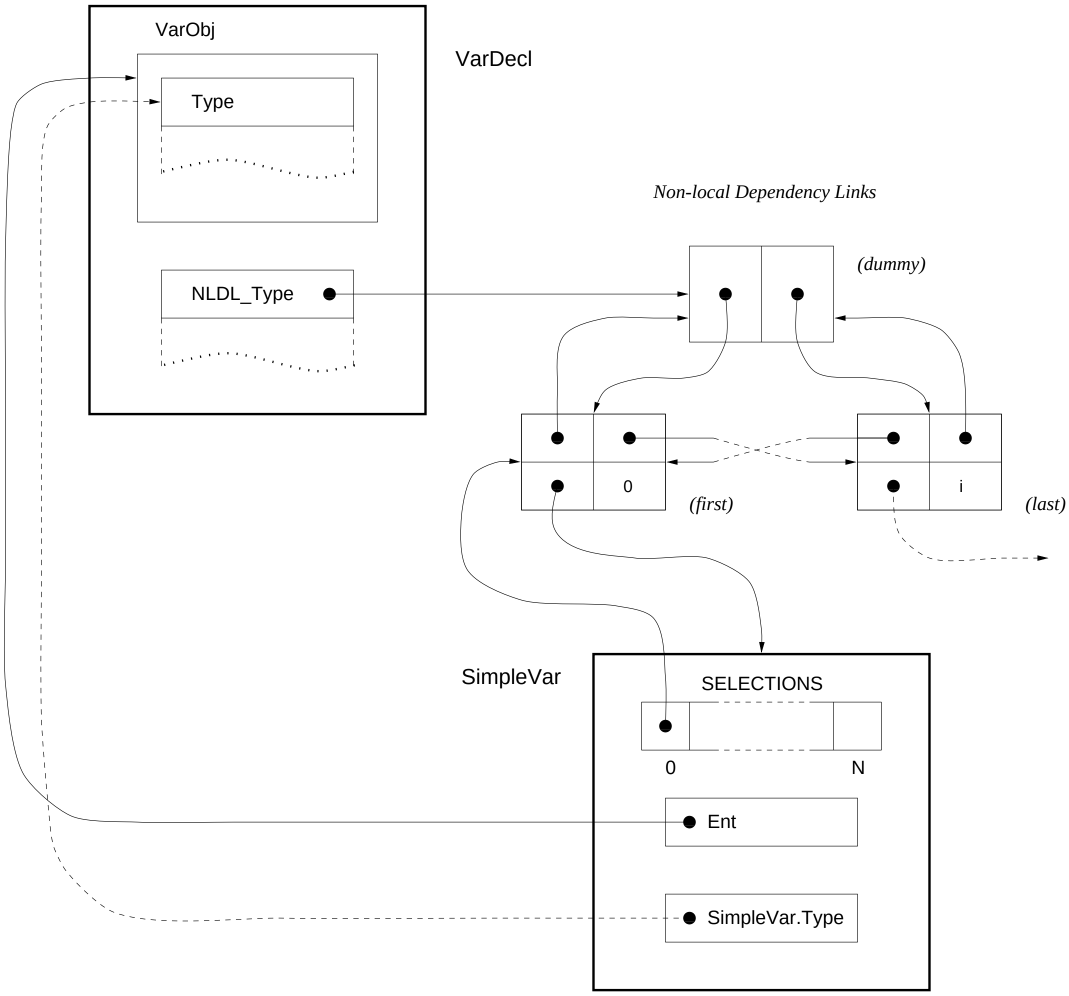


Figure 6.1: A variable declaration node and a simple variable reference in an **Example** program, showing the non-local dependency link from the `Type` component of the variable object to the variable reference node where it is accessed as a static selection.

```
method SimpleVar.VISIT_1(Ctx)
  ...
  % Evaluate a reference-valued attribute.
  Ent = ...
  % Register all selections from the attribute.
  % Call registration method with current node
  % and the selection index for the selection.
  % Here, we register selection of the 'Type' component.
  Ent.Add_NLD_Type(self, 0);
  ...
  % Once registered, the component may be accessed freely.
  return Ent.Type
  ...
```

Figure 6.2: Registering a static selection.

```
method VarObj.Add_NLD_Type(node, idx)
  % Create and initialize a new NLDL entry.
  nld = Create_NLD()
  nld.CONTEXT = node
  nld.SELECTION = idx
  % Insert new entry into NLDL for the 'Type' component.
  % The pseudovariable op denotes the AST node (operator) in which
  % the component and NLDL are located, which is captured in the closure
  % of this method.
  NLDL_Insert(nld, op.NLDL_Type)
  % Remove old entry from the NLDL that contains it.
  NLDL_Remove(node.SELECTIONS[idx])
  % Record current NLD link for this selection.
  node.SELECTIONS[idx] = nld
```

Figure 6.3: Registration method for `VarEntity.Type`.

```
method SimpleVar.DESTROY()
  ...
  % Notify all nodes containing selections from
  % a component in the deleted node.
  NLDL_Notify(self.NLDL_Type)
  ...
```

```
function NLDL_Notify(nldl)
  foreach elt in nldl do
    Notify_Change(elt.CONTEXT)
```

Figure 6.4: Handling NLDLs during node deletion.

Figure 6.5 shows the modifications required to implement this scheme. In the registration method, we create a new *non-local dependency* (NLD) link even if we know that the target of the old one has not been notified, because the selection base may have changed. The registration method correctly returns "notified" status on the initial evaluation, as the entries of the selection registration vector are initialized to **nil** when the AST node is created.

Objects can be allocated globally as well as within an AST node. Even though the components of global objects cannot change, constants of object type are often useful. Non-local dependency lists are not allocated for the components of global objects. Since it is not decidable in general whether a selection appearing in the attribute equations of an AST operator will access a global or a local object, the registration protocol must be uniform. We thus allocate a dummy registration method that does nothing, and returns **false** to indicate that the component has not changed.

6.5 Dynamic Component Selections

Dynamic selections pose additional difficulties because we do not necessarily know in advance how many distinct component instances may be selected by a single textual occurrence of such a selection. If the selection appears within a relation or a recursive function, an arbitrary number of instances may be examined over the course of its execution. We thus cannot allocate a selection registration vector as we did for static selections. Furthermore, dynamic selections may be performed within code that is textually remote from the constraint definitions for the node to which the dynamic dependency will be linked. For example, they may be embedded within globally-defined functions or relations, or within the methods of a class. We thus cannot statically associate these selections with the AST node visit functions in which they are invoked.

In the analyzer for **Example** developed in Chapter 4, declarations of named types (Figure 4.21) test for pathologically circular types by calling the function `CyclicType`:

```
Error(Name, "Cyclic type definition") :-
    CyclicType(TypeObj);
```

`CyclicType` in turn calls `CyclicTypeAux`, which contains the following rule:

```
method VarObj.Add_NLD_Type(node, idx)
  % Create and initialize a new NLDL entry.
  nld = Create_NLD()
  nld.CONTEXT = node
  nld.SELECTION = idx
  % Insert new entry into NLDL for the 'Type' component.
  NLDL_Insert(nld, op.NLDL_Type)
  if node.SELECTIONS[idx] == nil then
    node.SELECTIONS[idx] = nld
    % Selection has been previously notified.
    return true
  else
    % Remove old entry from the NLDL that contains it.
    NLDL_Remove(node.SELECTIONS[idx])
    % Record current NLD link for this selection.
    node.SELECTIONS[idx] = nld
    % Selection has not been previously notified.
    return false
```

```
method SimpleVar.VISIT_1(Ctx)
  ...
  Ent = ...
  if Ent.Add_NLD_Type(self, 0) then
    % 'Type' component has changed.
  else
    % 'Type' component is unchanged, unless the value
    % of attribute 'Ent' itself has changed.
  ...
  return Ent.Type
  ...
```

```
function NLDL_Notify(nldl)
  foreach elt in nldl do
    node = elt.CONTEXT
    idx = elt.SELECTION
    unless idx == nil then
      NLDL_Remove(elt)
      node.SELECTIONS[idx] = nil
    Notify_Change(node)
```

Figure 6.5: Revisions for registration method status return.

```
CyclicTypeAux(TsTYPENAME(Ent), Trail)
    => CyclicTypeAux(Ent.Type, [Ent|Trail]);
```

The dynamic dependency link created for the selection `Ent.Type` should be arranged such that the visit which originally called `CyclicType` will be retried if the value of the selected `Type` component changes.

We therefore maintain a *dynamic dependency context* (DDC), consisting of two global variables representing the node in which a visit function is currently executing, and the visit number for the visit in progress. When a child visit is performed, the old DDC is temporarily saved, and the DDC for the child visit is installed. Upon exit from the child visit, the saved DDC is restored. All dynamic selections occurring within the temporal scope of a DDC are "charged" to its associated AST node, which will result in notification of the node if the selected component subsequently changes value. A vector of *supports lists* is included in each node, one for each visit, which contain backlinks to the NLD links for all of the selections that support the node/visit pair, i.e., which were performed by the corresponding visit function. These data structures are illustrated in Figure 6.6.

When visiting a node, the evaluator will first retract any NLD links that were established the last time the visit was performed. The body of the visit function, including all subsidiary computations, will then be performed anew, and new NLD links will be established and added to the appropriate supports list. Such computations may be quite expensive, in which case a cache should be maintained for one or more of the functions or relations involved. Caching the results of function calls and the extension of relations is intimately connected with the full DDC protocol, which is described in the following chapter. In Figures 6.7 and 6.8, we extend the selection registration protocol to encompass dynamic as well as static selections. The same registration method is used in both cases, with a selection index of **nil** signifying a dynamic selection. Since the current node is now available from the DDC, it need no longer be provided as an argument, even in the case of a static selection.

In comparison to static selections, our algorithm provides less precise change notification for dynamic selections, as it does not attempt to account for which selections within a node are actually affected. In most cases, however, the evaluator is not prepared to exploit the missing information, as it re-evaluates attributes at the granularity of entire AST nodes. The special treatment of static selections does play an important role, as will soon become apparent. In the following chapter, we will extend the DDC and supports lists mechanism to accommodate the maintenance of caches for functions and relations.

6.6 Related Work

Christiansen [12] proposed that side-effects be used to simultaneously update the value of object (record) components at multiple remote attribute occurrences. He describes a static analysis that exploits structure-sharing to generate a more efficient static evaluator. He mentions the possibility of refining the static dependencies with dynamic traces, but does not pursue the idea further. Hedin [33] originated the description of long-distance dependencies by explicit linkage of objects embedded in the AST, as well as the use of functionally-attributed objects to model aggregates. In her approach, classes of objects are defined external to the AG formalism in hand-written code and are responsible for tracking the dynamic dependencies in which they participate. Given programmer-supplied declarations, the instances of the classes can be incorporated into an AG in a semi-automatic way, similar to the approach of Beshers and Campbell [6].

Boyland [10] permits remote access to the attributes of a node via a node reference, as if the entire node were itself an object in our sense. For example, rather than using an object to represent a declared entity, Boyland uses the entire attributed AST node for the declaration. We believe that

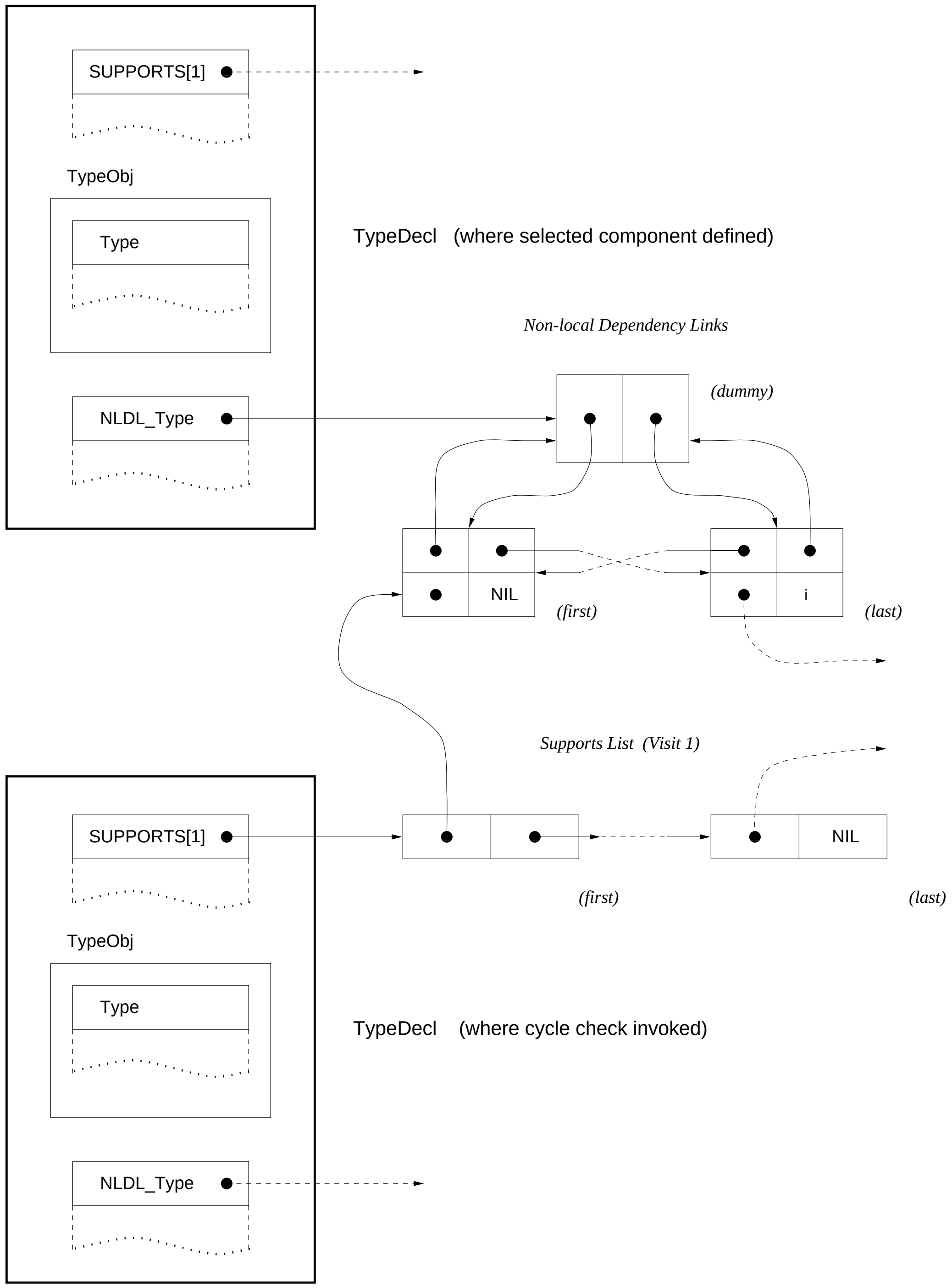


Figure 6.6: Two type declaration nodes in an **Example** program, showing the non-local dependency link from the `Type` component of a named type object to a node where it is accessed by the function `IsCyclic` as a dynamic selection.

```
% These global variable hold the current dynamic dependency context.

global *ddc-context*        % Node currently being visited.
global *ddc-visit*          % Visit number of current visit.


method TypeDecl.VISIT_4(Ctx)
  ...
  Retract_Supports(self.SUPPORTS[4])
  ...
  % Dynamically-bind DDC variables to node and visit
  % during evaluation of attributes.
  let *ddc-context* = self and *ddc-visit* = 4 in
    ...
    % Call out-of-line boolean-valued function.
    ... CyclicType(self.TypeObj) ...
    ...
  ...


function CyclicType(Ty)
  ...
  CyclicTypeAux(Ty, nil)
  ...


function CyclicTypeAux(Ty, Trail)
    ...
    % Register dynamic selection of 'Type'.
    Ent.Add_NLD_Type(nil)
    % Access component 'Type'.
    ... Ent.Type ...
    ...
```

Figure 6.7: Handling NLDLs with both static and dynamic selections.

```
method VarEntity.Add_NLD_Type(idx)
  node = *ddc-context*
  nld = Create_NLD()
  nld.CONTEXT = node
  nld.SELECTION = idx
  NLDL_Insert(nld, op.NLDL_Type)
  if idx == nil then % Registering dynamic selection.
    visit = *ddc-visit*
    % Add NLD link to supports list for this node/visit.
    node.SUPPORTS[visit] = cons(nld, node.SUPPORTS[visit])
    % Result is ignored for dynamic registrations.
    return false
  else % Registering static selection.
    if node.SELECTIONS[idx] == nil then
      node.SELECTIONS[idx] = nld
      return true
    else
      NLDL_Remove(node.SELECTIONS[idx])
      node.SELECTIONS[idx] = nld
      return false
```

```
function Retract_Supports(supps)
  foreach backlink in supps do
    NLDL_Remove(backlink)
```

```
function NLDL_Notify(nldl)
  foreach elt in nldl do
    node = elt.CONTEXT
    idx = elt.SELECTION
    unless idx == nil then
      % Static selection.
      NLDL_Remove(elt)
      node.SELECTIONS[idx] = nil
    Notify_Change(node)
```

Figure 6.8: Handling NLDLs with both static and dynamic selections (continued).

our separation of concerns between node references and objects is cleaner than Boyland's approach, particularly with respect to typing issues. In particular, we allow multiple object instantiations occurring within different AST operators to produce objects of the same type. Our work is also distinguished from that of Boyland in that we construct a statically-scheduled incremental evaluator, whereas his evaluator is dynamically-scheduled and must always perform a full re-evaluation.

Chapter 7

Functional Attributes

Functions in ADL may be declared with global scope or they may appear within the definition of a class or an AST operator, in which case their visibility is restricted to that context. Functions may also appear as attributes of AST nodes and objects. Functional attributes are a novel feature of ADL.

Mathematically, a function is a potentially infinite set of ordered pairs associating an argument value (typically itself a tuple) with a result value. Operationally, the concrete representations of computable functions provide a procedure for generating the members of the set on demand, yielding the associated result value when presented with a tuple of argument values. Functional attributes can thus compactly code a mapping that would take a large (or even unbounded) amount of space to represent as a first-order attribute value, perhaps as a list of tuples. A functional representation is particularly efficient when the value of the functional attribute is only sparsely observed, i.e., when a relatively small number of associations are actually demanded from each instance of the attribute. Every invocation of a function observes a single argument/result mapping of that function and some finite number of such mappings from functions invoked within subsidiary function calls. Dependencies involving function calls may be tracked at a fine level of granularity by tracking precisely which mappings were actually observed in computing the dependent results.

7.1 Representation of Aggregate Attributes

Functional attributes may be used to model aggregates. The ability to generate portions of an attribute value on demand, and to maintain precise dependencies based upon those parts actually observed, gives us a way to perform partial updates of aggregate attributes efficiently.

In the analyzer for **Example**, the binding environment is propagated to nearly every AST node in the program, and contains a binding for every declared name. Clearly, it would be very costly to update every instance of the binding environment when a single declaration is modified. By caching the results of calls to functional attributes, however, we can obtain an efficient solution in which individual bindings can be updated independently of other unchanged bindings. The binding environment is conveniently represented as a tree of binding contours providing lookup methods that can be queried as needed. Calls to these methods are cached, and dynamic dependencies maintained between the cache entries, which represent the argument/result mappings observed by the calls. We recall a few key points here; refer to Chapter 4 for the details.

The bindings introduced in a single group of declarations are collected in a binding contour, represented as an object of the `Contour` class. In **Example**, all contour objects except for the outermost are generated by the object instantiation **BodyCtx** in AST operator **Block**. Each contour except for the outermost contains a reference to another contour which represents its enclosing scope. This parent chain represents the hierarchical nesting of contours within the scoping discipline. A binding environment is represented as a reference to the innermost contour providing bindings to that environment. The **Contour** class provides a method **VisibleBinding**, which returns the innermost binding for the given identifier:

```
function  LocalBinding(String) -> BindingStatus;
```

The result is a reference to an object of the **Entity** class if a binding exists, or the special value **Undeclared** if one does not. The **VisibleBinding** method looks first for a binding locally within the contour, and if none is found, it delegates to the surrounding contour.

```
VisibleBinding(Ident) => Ent
    :- LocalBinding(Ident) => Ent & Ent /= Unknown;
VisibleBinding(Ident) => Ent
    :- Parent.VisibleBinding(Ident) => Ent;
```

The search for the local binding succeeds if a unique binding is found in the local binding relation **Binds**.

```
LocalBinding(Ident) => Ent
    :- Binds(Ident, Ent) & ~Duplicate(Ident, Ent);
LocalBinding(Ident) => Unknown;
```

The tuples of **Binds** are collected as the value of a relational attribute, propagated to the instantiation of the contour object from the declarations. A name is resolved within a binding environment by invoking the **VisibleBinding** method, such as within a reference to a variable:

```
attribute Ent : BindingStatus =
    SimpleVar.Ctx.VisibleBinding(Name.Text);
```

The name resolution above depends only on the identity of the contour representing the binding environment and value of **VisibleBinding** (as a set of ordered pairs) at a single domain value. This result depends in turn on the value of **LocalBinding** at a single domain value, and possibly the **VisibleBinding** value at the parent contour. We specify that caches be provided for **VisibleBinding** and its auxiliary functions **LocalBinding** and **Duplicate**. Changes are propagated only to mappings that actually depend on them, and propagation stops as soon as possible. Leaf calls to **VisibleBinding**, i.e., at the nodes where identifier uses occur, are notified only if the binding to which they refer has changed.

7.2 Embedding of Functions in the AST

The body of a function definition may reference names other than the arguments. In ADL, such names must necessarily be those of other attributes, as the definition of a function is not permitted syntactically within the scope of a variable. The collection of attributes referenced within the body of a function is called its *closure*, by analogy with the use of that term in the context of conventional functional programming languages. Since functional attributes are not first-class values, however, explicit closure objects need not be constructed at runtime. Any attributes referenced within the body of a functional attribute are simply stored in the surrounding tree node, which is available to the function during execution as an implicit argument.

When a functional attribute of an adjacent AST node is invoked, the calling site knows only the phylum to which the node belongs. The actual function body that is executed depends on the AST operator of which the node is an instance. Since our AGs are in normal form, every functional attribute occurrence (except for local attributes) may be either defined or invoked, but not both. The synthesized functional attributes of an AST node and the inherited functional occurrences of its children must be provided with function bodies which can then be invoked from the complementary attributes of adjacent nodes.

Synthesized functional attributes are handled easily when the target language is object-oriented. We simply declare each functional attribute as a virtual method of a class associated with each phylum and inherited by the class associated with each AST operator. In the operator class, we override the virtual method with the function body appropriate for nodes labeled with the operator. Inherited functional attributes are more complicated, as the correct function body depends not only upon the calling node's parent, but also upon which of the parent's children it happens to be. Techniques for solving this problem, which arises in the construction of demand-driven evaluators for conventional AGs, are well known, and are described in Engelfriet [18] and Jourdan [41]. These methods are awkward when rendered in a strongly typed object-oriented target language, however. For this reason, we restrict functional attributes to be synthesized only. It is possible to achieve the effect of inherited functional attributes using objects, suggesting that the technique could be automated in order to relax this restriction.

Definition of recursive functions as functional attributes gives rise to circular attribute dependencies. Since every function definition, however, is technically an attribute definition, this would seem to prohibit recursive functions entirely! By special dispensation, we permit mutually-recursive definition of one or more *local* attributes appearing within the same AST operator or at the top-level of the language description. Cliques of such mutually-recursive definitions are treated as a unit during attribute dependency analysis; i.e., every attribute in the clique is taken to depend on the predecessors of all of the others, except for those that are themselves members of the clique.

Functional attributes occur most often as components of objects, usually inherited from the class definition. Such attributes function in every way as local attributes of the surrounding AST node as well as being remotely-accessible via references to the object. All of the functional attributes appearing as examples in this chapter are components of `Contour` objects. We will sketch the handling of functional attributes accessible from adjacent nodes, but the details will be presented in the following chapter in the context of relational attributes.

7.3 Caching Function Calls

Incremental evaluation relies on the retention and reuse of the results of computations performed previously. A functional attribute implicitly defines an argument/result mapping, a potentially unbounded set of argument-tuple/result pairs. While the value of a function at a given argument tuple can always be recomputed on demand, the fine-grained dependency tracking discussed earlier requires that the mapping be explicitly represented. Usually, only a portion of the mapping will actually be observed via any given function call, or even the totality of function calls within the execution of the analysis. We construct a partial representation of the mapping by caching the results of function calls as they occur. Each entry in the cache will be called a *mapping*, not to be confused with the abstract mapping that the function represents. Entries must be removed from the cache when they are no longer valid. Our algorithm keeps track of mappings that may have changed, and retries them to determine if they are still valid. A function for which a cache is provided is called a *maintained function.*[1]

The value returned by a function call depends not only on the arguments, but also upon attributes referenced within the body of the function (i.e., the closure) and by the values returned by subordinate function calls. For a given tuple of arguments, the value returned by a call to a functional attribute may change when any of the following change:

- The value returned by a subordinate function call.
- The value returned by a subordinate component selection.
- The value of a scalar or relational attribute referenced by the function or by a function that it calls transitively.
- The value of a child pointer by which a functional attribute is accessed during the call.

Thus, whenever we cache the result of a function call, we must arrange for the call to be retried when any of these conditions obtains. This will result in the propagation of further changes should the cached mapping be found invalid.

We reduce storage costs by recording dependencies statically wherever possible. We associate an implicit boolean *status attribute* with each functional attribute. The status attribute is treated similarly to an ordinary scalar attribute, and takes on a **true** value at any occurrence where the underlying functional attribute must be considered changed. The status attributes, computed during the statically scheduled visit procedures, allow us to reconcile our eager, statically-scheduled change propagation strategy with the demand-driven behavior of function calls and caching. The status attribute for a functional attribute is **true**, indicating change, if:

- The value of a scalar attribute (including object references) referenced in the function body has changed.
- A selection of an object component via a reference-valued attribute has been notified, indicating that the value of the component has changed.
- The status attribute of a functional (or relational) attribute called within the function body has the value **true** indicating that the call may possibly no longer return the same result.
- The identity of the node from which the status value for another functional (or relational) attribute was received has changed due to a modification to the structure of the tree.

In the absence of caching, the status value reflects whether the *closure* of the function, including its transitive dependencies, has changed. This is a necessary (though not sufficient) condition for the value of the function at a given set of arguments to change. We must conservatively assume that any call to a function whose status is **true** must be retried; thus any visit procedure receiving a **true** status attribute as an argument must be executed. All scalar attributes that depend on the function will be re-evaluated, retrying the calls to the functional attribute in the process. Unlike ordinary scalar attributes, the status value is not compared against a previously saved value. Its value during the traversal indicates change status directly.

The scheme described so far is unrealistically pessimistic in that it will re-evaluate every call to a functional attribute that is transitively dependent upon a changed scalar attribute. By caching the

[1]This term is used by Hoover [37] in exactly the same sense as we use it here. We chose it in analogy to the "maintained properties" of Ballance's Colander system [4], which inspired many aspects of the present work.

argument/result pairs observed at each function attribute, it can be determined when the observed mapping has *actually* changed, and thereby the status value for the attribute can be refined. First the "incoming" status is computed as before. Then, if it indicates that the function may have changed, the saved argument/result pairs are retried. The "outgoing" status indicates a change only if one or more argument tuples was mapped to a different result. It is not necessary to provide a cache for every function. The scheme remains correct if caching is omitted for some of the attributes, but more function calls may be re-evaluated (and more nodes visited) than would otherwise be required.

As developed so far, our caching scheme can stop change propagation when it discovers that a function is completely unchanged – i.e., implements the same argument/result mapping – but must still examine every successor should any changes be noted. We still need one further refinement in order to get the fine-grained dependency tracking that we seek. This caching scheme fails to take account of the fact that a call with a specific argument tuple will observe the value of other functional attributes only at certain argument values. If the subsidiary call has been cached, the current call should depend only on the cached argument/result pairs that were observed. Our refined algorithm thus records dynamic dependency links, similar to those used to link selections and object components, between each cached mapping and the mappings that were examined during the subsidiary calls. By recording such fine-grained dependencies, effectively "splitting" an attribute dynamically into as many dependency traces as observed argument tuples, aggregates can be represented efficiently using functional attributes. Dynamic component selections occurring within a function call are not part of the closure, and will not be accounted for by the status attribute. These selections, however, will create dynamic dependency links in the same way as subsidiary function calls.

Since not all functions have caches, a call to a subsidiary maintained function or a dynamic component selection may be charged to a function other than the one that performed it, namely, the innermost maintained function on the dynamic call chain. If no such function exists, the dependencies are charged to the AST node visit which contains the outermost call.

In this chapter, we will be dealing with functional attributes that are methods of an object, and are thus local attributes. Much of the handling of the status attributes described above is only applicable to attributes that may be invoked from adjacent nodes. In the remainder of this chapter, we will not discuss these issues in detail. Analogous issues arise in the handling of relational attributes, and will be covered in the following chapter.

7.4 Implementing Maintained Functions

The cache for a function is represented in the compiled analysis description as an object in the target language.[2] The slots of the object include:

- A pointer back to the AST node containing the cache.
- A *retry list* containing mappings that have been marked potentially invalid.
- A hash table containing *mappings*, indexed by the argument tuple.

Each mapping in the hash table contains:

- A pointer back to the cache to which the mapping belongs.

[2] Actually, we use a Common Lisp `defstruct` with a few functional components for the methods. This is an inessential implementation detail that will be ignored here.

- A *successor list*, containing a doubly-linked list of *dynamic successor link* (DSL) records, analogous to the NLD links of the previous chapter. Each DSL record has a pointer back to the mapping to which it belongs, and a pointer to an AST node or another mapping that constitutes a dependency successor for the current mapping.
- A list of *backlinks* pointing to DSL records that refer to this mapping as their successors.

This structure is illustrated in Figure 7.1, which shows two function caches, each with a "typical" mapping record. In the figure, the mapping shown in the upper cache is a dependency successor to that shown in the lower one. The longer dashed lines linking the two caches represent the DSL successor pointer and its corresponding backlink.

Calls to a maintained function are handled by the `EVAL` method of the cache. The uncached semantics of the function are implemented by the `FILL` method. The `EVAL` method examines the cache to see if it contains a mapping for the arguments provided in the call. If so, the cached result is returned, else the `FILL` method is called to compute the result, and a new mapping is installed in the cache by the `EVAL` method before it returns. The `EVAL` method for `BodyCtx.VisibleBinding` is shown in Figure 7.2.

A mapping must be updated or removed from the cache if the cached result no longer agrees with that which would be yielded by the `FILL` method when applied to the argument values saved with the cache entry. Cache entries whose validity have been called into question due to a change in the value of a dependency predecessor are retried and updated as necessary. Cache entries that are no longer observed, i.e., that no longer contribute to the value of any attribute, are removed from the cache entirely.

Because caching may be selectively enabled, the protocol for calling a function must remain the same whether a cache is provided or not. In actuality, another wrapper function is used, which is not a method of the function cache object, and which calls the `EVAL` method. In the case of a functional attribute without a cache, the wrapper function simply implements the basic semantics of the function as the `FILL` method would. This detail is glossed in the pseudocode presented in this chapter.

7.5 Calling a Maintained Function

Every functional attribute call takes place within a *dynamic dependency context* (DDC) as described in the previous chapter. Previously, the DDC was always an AST node/visit number pair. We now extend the definition of a DDC to consist of either a node/visit pair or a mapping (function cache entry). Dynamic dependencies are created not only for selections, but also for calls to maintained functions. When each dynamic dependency is created, its dependency successor will be available in the DDC, as either a node/visit or a cached function call.

When a maintained function is called, a *dynamic successor link* (DSL) pointing to the current DDC is added to the argument/result mapping accessed, and a backpointer is added to the DDC. If a new cache entry had to be allocated to record the mapping, its argument slots are filled in and then the body of the maintained function is performed using the new mapping as the DDC.

Upon entry to a visit, all dynamic dependencies supporting the visit are retracted. All of the attribute evaluations scheduled for the visit are then performed again, re-establishing the correct supports for the node and visit. It is possible, however, that a mapping that previously supported the current node/visit is no longer needed. If the mapping no longer supports *any* context, it should be reclaimed. A scan is thus performed over the original supports list upon exit from the visit, and mappings with no successors are deleted. Figure 7.3 illustrates the handling of a call to a maintained function from within an AST node visit.

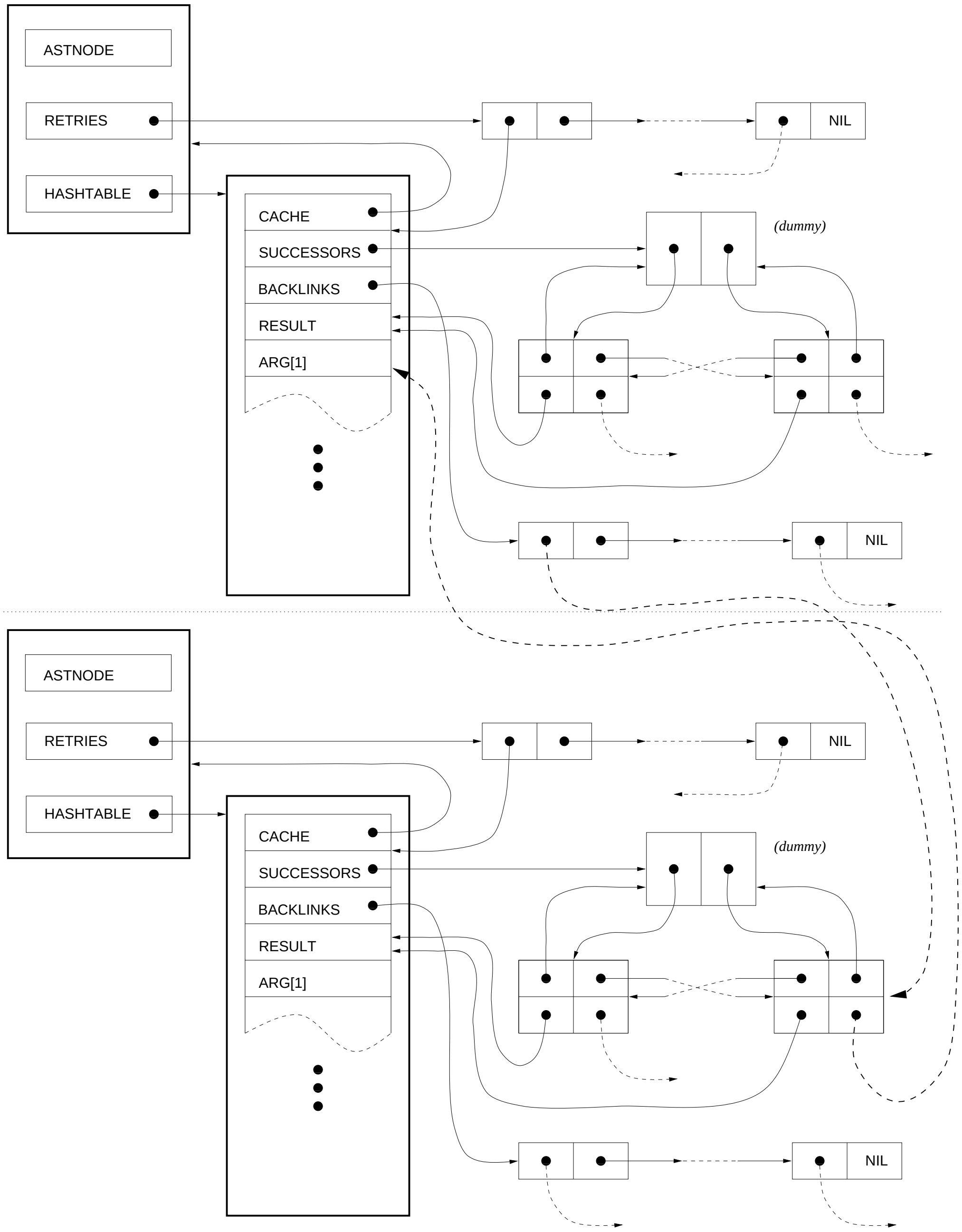


Figure 7.1: The cache data structure for maintained functions. Two caches are shown, in which the upper cache contains a mapping that supports another mapping in the lower cache.

```
method BodyCtx.VisibleBinding.EVAL(Ident)
  table = self.HASHTABLE
  entry = Hash_Lookup(table, [Ident])
  if entry /= nil then % Cache hit.
    % Make current DDC depend on the mapping, so that it
    % will be notified if the cached mapping is invalidated.
    Add_Dynamic_Predecessor_Mapping(entry)
    % Return the cached result.
    return entry.RESULT
  else % Cache miss.
    mapping = Create_Mapping()
    % Point back to cache containing mapping.
    mapping.CACHE = self
    % Store arguments.
    mapping.ARGS = [Ident]
    % Make current DDC depend on the mapping, so that it
    % will be notified if the cached mapping is invalidated.
    Add_Dynamic_Predecessor_Mapping(mapping)
    % Compute the result, using the new mapping as DDC.
    let *ddc-context* = mapping in
      % The 'FILL' method actually computes the result.
      result = self.FILL(Ident)
      mapping.RESULT = result
      % Enter the new mapping in the cache.
      Hash_Enter(table, [Ident], mapping)
      return result
```

```
function Add_Dynamic_Predecessor_Mapping(mapping)
  context = *ddc-context*
  % Create new dynamic successor link.
  dsl = Create_DSL()
  dsl.SUCCESSOR = context
  dsl.MAPPING = mapping
  % Insert new DSL into successor list of mapping.
  DSLL_Insert(dsl, mapping.SUCCESSORS)
  % Register backlink, depending on kind of DDC in effect.
  typecase context of
    operator:
      visit = *ddc-visit*
      context.SUPPORTS[visit] = cons(mapping, context.SUPPORTS[visit])
    mapping:
      context.BACKLINKS = cons(mapping, context.BACKLINKS)
```

Figure 7.2: Evaluation of a maintained function using a wrapper method.

```
global *saved-supports*  % Temporary stack using during AST traversal.


method SimpleVar.VISIT_1(Ctx)
  ...
  % Remove all dependencies supporting this visit.
  Stack_Push(self.SUPPORTS[1], *saved-supports*)
  Retract_Support(self.SUPPORTS[1])
  self.SUPPORTS[1] = nil
  ...
  % Dynamically-bind DDC variables to node and visit
  % during evaluation of attributes.
  let *ddc-context* = self and *ddc-visit* = 1 in
    % Evaluate attributes here.
    ...
    % Invoke maintained function (method of remote object).
    ... Ctx.VisibleBinding.EVAL(TextOf(self.Name)) ...
    ...
  % Clean up 'stale' dependencies.
  Commit_Retractions(Stack_Pop(*saved-supports*))
  return ...


function Retract_Support(backlinks)
  % Remove all dynamic dependencies that support
  % the context to which 'backlinks' belong.
  foreach link in backlinks do
    typecase link of
      'nld': NLDL_Remove(link)
      'dsl': DSLL_Remove(link)


function Commit_Retractions(backlinks)
  % Delete 'stale' mappings, i.e., those that do not
  % currently support any DDC.
  foreach link in backlinks do
    when link istype 'dsl' then
      mapping = link.MAPPING
      when Empty_DSLL(mapping.SUCCESSORS) then
        Delete_Mapping(mapping)
```

Figure 7.3: Calling a maintained function during a node visit.

7.6 Updating the Caches

Static dependencies involving functional attributes are approximated statically in the same manner as any other kind of attribute. Thus each functional attribute is scheduled for evaluation during some visit to the node to which it belongs. Furthermore, it is guaranteed that any other attribute upon which the functional attribute might depend has been scheduled earlier in the AST traversal.

When the visit-sequence evaluator reaches a point in its traversal where the evaluation of a functional attribute is scheduled, the cached function calls that have potentially changed are retried, and the saved argument/result associations updated if necessary. If the incoming status is **true**, every mapping in the cache must be retried. Otherwise, only those mappings that have been placed on its *retry list* by an earlier change notification are retried. If any cached association is found to have changed, its dynamic successors are then notified. The outgoing status for a maintained function is always **false** as more precise change information is propagated to its successors via the dynamic notification mechanism.

During retry, the supports for the mapping are retracted. They will be restored if still in effect during the retry. After the retry is complete, the original supports are examined again, and any mappings that are no longer needed are removed. This process is entirely analogous to the handling of the supports list during a node visit, reflecting the fact that node visits and mappings represent the two possibilities for the DDC.

Figure 7.4 illustrates the code generated for the evaluation (cache update) of the `BodyCtx.VisibleBinding` function during a visit to a `Block` node. Most of this code reflects the computation of the status attribute. The actual cache update is performed by the runtime support routines in Figure 7.5.

If a successor DDC to be notified is itself another cache entry, it is placed on the retry list of the function cache to which it belongs, and the containing node is marked for visits. Otherwise, the DDC is an AST node, which is simply marked for visits. In either case, every call that observed the changed association will eventually be retried at its normal place in the evaluation schedule as determined by the static dependency analysis and encoded in the visit procedures. Figure 7.6 shows the notification procedure.

Caching of function calls carries a cost in both space and time, and thus should be used only where significant savings due to incremental evaluation are observed. The use of caching is optional, and is enabled by the author of the language description where appropriate. The caching scheme described here allows maintained and non-maintained functions to be mixed freely. Any dynamic dependencies that cannot be charged to the immediately enclosing function call because it was not cached are simply charged to maintained functions higher on the call chain, or to the node/visit that initiated the outermost call. There is one special case that requires care. In Figure 7.4, we could omit propagating the status value for a functional component to the remote call sites using the NLDL mechanism, because the outgoing status value for a maintained function is always **false**. In the case of a non-maintained function, the outgoing status value is significant, and must be handled in the normal manner. Figure 7.7 shows the code that we would have generated had no cache been provided for `BodyCtx.VisibleBinding`. Because a remote call site does not know whether a functional component has been provided with a cache or not, it will always attempt to register a non-local dependency. In the case of a non-maintained functional component, where we do not create an NLDL, the registration method is a dummy which does nothing but return **false**.

```
method Block.VISIT_1(Ctx)
  ...
  Stack_Push(self.SUPPORTS[1], *saved-supports*)
  Retract_Support(self.SUPPORTS[1])
  self.SUPPORTS[1] = nil
  ...
  % Dynamically-bind DDC variables to node and visit
  % during evaluation of attributes.
  let *ddc-context* = self and *ddc-visit* = 1 in
    ...
    LocalBinding_STATUS = ...
    ...
    % Attribute 'Parent' is in the closure for 'BodyCtx.VisibleBinding'
    % and must therefore be saved within the AST.
    parent = Ctx
    self.BodyCtx_VisibleBinding_CHANGED = false
    when parent.Add_NLD_VisibleBinding(0) then
      % Status changed for 'Parent.VisibleBinding', upon which
      % 'BodyCtx.VisibleBinding' depends.
      self.BodyCtx_VisibleBinding_CHANGED = true
    unless parent == self.BodyCtx.Parent then
      % If the new parent does not match the one currently saved as a
      % component of 'BodyCtx', update it.
      self.BodyCtx.Parent = parent
      % Since 'BodyCtx.VisibleBinding' references 'BodyCtx.Parent',
      % we must now mark its closure as changed.
      self.BodyCtx_VisibleBinding_CHANGED = true
      % Notify any context that examines 'BodyCtx.Parent' remotely.
      NLDL_Notify(self.NLDL_Parent)
    VisibleBinding_STATUS =
        % The status of 'BodyCtx.VisibleBinding' depends on that
        % of any functions or relations that it references, as well as
        % any ordinary (scalar) attributes.
        LocalBinding_STATUS and self.BodyCtx_VisibleBinding_CHANGED
     % Update the cache, using the status to optimize.
     % It is not necessary to notify 'NLDL_VisibleBinding', as any remote
     % call sites accessing a changed mapping will be notified during
     % update.
     Retry_Cached_Mappings(self.VisibleBinding, VisibleBinding_STATUS)
    ...
  Commit_Retractions(Stack_Pop(*saved-supports*))
  return ...
```

Figure 7.4: Evaluating (updating) a maintained functional attribute.

```
global *cache-changed*

function Retry_Cached_Mappings(cache, status)
  *cache-changed* = false
  if status == false then  % Closure has not changed.
    % Retry notified mappings only.
    retries = cache.RETRIES
    unless retries == nil then
      foreach mapping in retries do
        Retry_Mapping(cache, mapping)
      cache.RETRIES = nil
    return *cache-changed*
  else  % Closure has changed.
    % All mappings are potentially invalid.  Retry all.
    foreach mapping in cache.HASHTABLE do
      Retry_Mapping(cache, mapping)
    cache.RETRIES = nil
    return *cache-changed*

function Retry_Mapping(cache, mapping)
  % The retried computations support the mapping being
  % retried, therefore the retried mapping is the DDC.
  let *ddc-context* = mapping in
    % Remove all successor pointers to this node from our predecessors.
    backlinks = mapping.BACKLINKS
    mapping.BACKLINKS = nil
    Retract_Support(backlinks)
    % Retry function call.
    result = cache.FILL(mapping.ARGS)
    unless result == mapping.RESULT then % Result changed.
      % Update cached mapping and notify successors.
      mapping.RESULT = result
      *cache-changed* = true
      Notify_Mapping_Successors(mapping)
    Commit_Retractions(backlinks)
```

Figure 7.5: Retrying the mappings for a maintained function.

```
function Notify_Mapping_Successors(mapping)
  % Force retry of dependent DDC.
  foreach succ in mapping.SUCCESSORS do
    typecase succ of
      operator:
        % Force visit to node.
        Notify_Change(succ)
      mapping:
        cache = succ.CACHE
        % Put mapping on retry list for cache.
        cache.RETRIES = cons(succ, cache.RETRIES)
        % Force visit to node so cache will be updated.
        Notify_Change(cache.ASTNODE)
```

Figure 7.6: Notifying the DDC when a mapping changes.

7.7 Loose Ends and Housekeeping

Allowing the DDC to consist of a mapping record requires a small change to the code for tracking component selection dependencies (NLDs), as the DDC for the selection may be a mapping instead of a node/visit. Dynamic selections (changes to which are not reflected in the status attributes) from within both functional attributes and ordinary semantic functions generate dynamic dependencies, and are treated similarly to calls of maintained functional attributes. When called with a null index value, selection registration methods add a DSL to the non-local dependency list for the selected component, pointing to the current DDC. Notification and retry proceed as for calls to functional attributes. The revised versions of the selection registration method and the notification procedure are shown in Figure 7.8.

When a mapping is deleted from its cache, all of its successors must be notified. When a node containing the cache for a function is destroyed, all of the successors of the mappings it contains must be notified, though we can simply abandon the entire cache and need not actually remove its mappings from the hash table. When a node is destroyed, there may still be cache entries within the AST supported by its mappings. This will be corrected during the AST traversal, but may result in the attempted redundant deletion of those mappings. A reserved value, denoted here as `"DELETED"`, is stored in the backlinks slot to mark deleted mappings and thus suppress duplicate deletions. Figure 7.9 shows the code for handling the deletion of mappings, including destruction of the cache when its containing node is removed from the AST.

While we call them “attributes” in ADL, entities declared at top-level in an analysis description, and which are not a part of an AST node, are not true attributes in the formal sense of attribute grammars. Since they are evaluated only once at initialization time, we cannot apply the cache maintenance protocol described here. Caches are thus never provided for global functions.

7.8 Related Work

Vogt, Swierstra, and Kuiper [75] have applied caching to the visit functions for evaluating OAGs in order to build an incremental evaluator for classical AGs in a functional setting. They extend their

```
method Block.VISIT_1(Ctx)
  ...
  Stack_Push(self.SUPPORTS[1], *saved-supports*)
  Retract_Support(self.SUPPORTS[1])
  self.SUPPORTS[1] = nil
  ...
  let *ddc-context* = self and *ddc-visit* = 1 in
    ...
    LocalBinding_STATUS = ...
    ...
    parent = Ctx
    self.BodyCtx_VisibleBinding_CHANGED = false
    when parent.Add_NLD_VisibleBinding(0) then
      self.BodyCtx_VisibleBinding_CHANGED = true
    unless parent == self.BodyCtx.Parent then
      self.BodyCtx.Parent = parent
      self.BodyCtx_VisibleBinding_CHANGED = true
      % Notify any context that examines 'BodyCtx.Parent' remotely.
      NLDL_Notify(self.NLDL_Parent)
    VisibleBinding_STATUS =
        LocalBinding_STATUS and self.BodyCtx_VisibleBinding_CHANGED
    % As there is no cache for 'Ctx.VisibleBinding', we cannot determine
    % if any mappings changed, nor can we notify nonlocal successors on
    % the basis of mappings actually observed. We thus conservatively
    % force a full invalidation of our immediate successors.
    when VisibleBinding_STATUS then
      NLDL_Notify(self.NLDL_VisibleBinding)
    ...
  Commit_Retractions(StackPop(*saved-supports*))
  return ...
```

Figure 7.7: Evaluating a non-maintained functional attribute.

```
method VarEntity.Add_NLD_Type(idx)
  context = *ddc-context*
  nld = create_NLD()
  nld.CONTEXT = context
  nld.SELECTION = idx
  NLDL_Insert(nld, op.NLDL_Type)
  if idx == nil then % Registering dynamic selection.
    typecase context of
      operator:
        visit = *ddc-visit*
        % Add NLD link to supports list for this node/visit.
        context.SUPPORTS[visit] = cons(nld, context.SUPPORTS[visit])
      mapping:
        context.BACKLINKS = cons(nld, context.BACKLINKS)
    % Result is ignored for dynamic registrations.
    return false
  else % Registering static selection.  Context must be AST node.
    if context.SELECTIONS[idx] == nil then
      context.SELECTIONS[idx] = nld
      return true
    else
      NLDL_Remove(context.SELECTIONS[idx])
      context.SELECTIONS[idx] = nld
      return false
```

```
function NLDL_Notify(nldl)
  foreach elt in nldl do
    context = elt.CONTEXT
    idx = elt.SELECTION
    typecase context of
      operator:
        unless idx == nil then
          % Static selection.
          NLDL_Remove(elt)
          context.SELECTIONS[idx] = nil
        Notify_Change(context)
      mapping:
        cache = context.CACHE
        cache.RETRIES = cons(context, cache.RETRIES)
        Notify_Change(cache.ASTNODE)
```

Figure 7.8: Revised handling of selection dependencies.

```
function Delete_Mapping(mapping)
  backlinks = mapping.BACKLINKS
  unless backlinks == "DELETED" then
    unless Empty_List(mapping.SUCCESSORS) then
      Notify_Mapping_Successors(mapping)
    Hash_Remove(mapping.CACHE.HASHTABLE, mapping.ARGS)
    mapping.BACKLINKS = "DELETED"


function Destroy_Function_Cache(cache)
  foreach mapping in cache do
    Retract_Support(mapping.BACKLINKS)
    mapping.BACKLINKS = "DELETED"
    Notify_Mapping_Successors(mapping)
```

Figure 7.9: Deletion of cached mappings.

method to include a variant of attribute grammars they call *higher-order attribute grammars* (HAGs), in which AST-valued attributes may themselves be recursively attributed. Higher-order attributes may be used to model functions, allowing the dynamic call tree to be reified as an explicit data structure, which, in conjunction with caching, permits re-use of computation in a manner similar to our use of maintained functional attributes. Swierstra and Vogt have explored this technique [71], including an application to incremental name resolution.

Pugh [61] explores function caching as a general technique for incremental computation. Although he discusses the application of function caching to classical incremental attribute evaluation, he is concerned principally with general recursive programs apart from attribute grammars. In common with Swierstra and Vogt, cache housekeeping is not adequately addressed, permitting the cache to contain stale entries, or entries that were flushed prematurely and must be unnecessarily regenerated.

Hoover's Alphonse program transformation system [37] pioneered the combination of topological-order change propagation and function caching. In response to user-supplied annotations, calls to functions and methods in an imperative object-oriented language can be cached to permit efficient incremental execution. Because Alphonse supports unrestricted imperative programming, all dependencies, including those between scalars, must be recorded dynamically in general, though Hoover suggests the use of flow analysis to exploit static dependencies opportunistically. The shape of the dependency graph is completely under program control and changes as the program executes, thus the highly-effective static dependency analysis methods applicable to attribute grammars cannot be used. Alphonse is forced to use a complex dynamic node-priority labeling algorithm in order to perform change propagation in topological order. Our approach to change propagation for maintained functional attributes is essentially the same as Hoover's, but exploits the extensive static analysis permitted by AGs. All dynamic dependencies in our system merely refine coarser dependencies used in generating the visit-procedures. We then can enumerate the function caches in topological order during the tree walk, skipping visits and cache updates when the absence of a recorded dynamic dependency warrants it.

Chapter 8

Relational Attributes and Collections

Conceptually, relations generalize functions by removing the *a priori* distinction between arguments and results. Relations may also be used to model functions in which an argument is mapped into multiple results. Operationally, however, our treatment of relations is weaker and less efficient than that of functions, making the use of functions preferable where possible. Both global relations and relational attributes are supported. Collections are an alternative realization of relations which are more convenient to use in many cases, and which trade off some generality for a highly-efficient implementation.

8.1 Representation of Sets using Relations

Relations in ADL are most appropriately used for modeling sets, as their operational semantics result in the complete enumeration of the set of tuples that constitute the relation, in contrast to the generation of tuples "on demand" in the case of functions. Effectively, the operational semantics assumes that all of the elements in the tuples of a relation are results, not arguments, even though element values that are known at the relation query site will probably be viewed conceptually as arguments by the description writer.

In the analyzer for **Example**, the bindings created locally within each scope are collected in the synthesized relational attribute `Binds` of declarations and declaration sequences:

```
relation Binds(String, Entity);
```

Initially, each declaration provides a single tuple to its `Binds` relation, representing the binding created by that declaration:

```
VarDecl.Binds(Var.Text, VarObj);
```

Here, the single rule for the relation `VarDecl.Binds` is an unguarded *unit clause* representing the binding of the string name of the variable to the `VarObj` entity that represents it. The **`analyze`** construct for the sequence of declarations decomposes the sequence recursively into smaller subsequences until only singleton declarations remain (Figure 4.22). The binding sets for the subsequences are then combined:

```
Decls.Binds(Ident, Ent) :- d1.Binds(Ident, Ent);
Decls.Binds(Ident, Ent) :- d2.Binds(Ident, Ent);
```

The resulting set of bindings is then installed in the `Contour` object of the surrounding `Block` statement:

```
object BodyCtx : NormalContour
where
  ...
  Binds(Ident, Ent) :- Decls.Binds(Ident, Ent);
end BodyCtx;
```

Relational attributes are implemented as generators, i.e., as procedures that repeatedly invoke a procedural argument with successive values of the tuples of the relation, arranged in an arbitrary order. We thus specify that the `Binds` component of the object be explicitly materialized, as it is more efficient to enumerate the tuples from such a representation than to generate them afresh upon each access.

8.2 Implementing Relations as Generators

Relational attributes are rendered in the target language as procedures, that is, a method with a null or unspecified result. These procedures are embedded within the AST in exactly the same manner as functional attributes. The procedure representing each relational attribute, called its *generator*, takes a single argument, called a *continuation*, which is itself a procedure with one argument for each column (i.e., argument) of the relation. A relation is queried by invoking its generator with a continuation that is prepared to perform the processing desired on each tuple yielded by the generator. The generator for `VarDecl.Binds` simply invokes its continuation on the pair of arguments consisting of the textual name of the child `Var` (an identifier lexeme) and the object `VarObj`:

```
method VarDecl.Binds(CONT)
  CONT(TextOf(self.Var), self.VarObj)
```

The arguments are simply passed on through the internal sequence nodes belonging to the `analyze` construct within AST operator `Declarations`. The interesting case is the pairing operator:

```
method DeclList_Decls_PAIR.Binds(CONT)
  self.LEFT.Binds(lambda(x, y){ CONT(x, y) })
  self.RIGHT.Binds(lambda(x, y){ CONT(x, y) })
```

The resulting relation is then passed on to the `DeclList` operator itself with a simple copy rule:[1]

```
method DeclList.Binds(CONT)
  self.Decls.Binds(lambda(x, y){ CONT(x, y) })
```

Like functional attributes, relational attributes must be synthesized. Inherited relational attributes are not permitted. This restriction arises from the same implementation concerns as in the functional case.

[1] The expression **lambda**`(x, y){ CONT(x, y) }` can be simplified to `CONT` using the familiar rule of η-reduction. Our implementation does not do this, however.

8.3 Maintained Relations

Generating a relation afresh upon each occasion where it is queried may be very expensive. Furthermore, the evaluator cannot easily determine if a relation value has changed, thus every immediate successor to a relation must be re-evaluated upon every incremental re-analysis. To avoid these costs, relations may be cached. We call such relations *maintained relations*. The implementation of maintained relations is similar to that of maintained functions, but is simplified somewhat by the fact that the entire relation is cached as a unit. Since tuple-level dependencies among relations are not recorded, partial update is not possible. The entire relation cache then functions in a role similar to the mapping records used in the functional case, as if there were a single such record to represent the totality of the relation.

A relation cache is represented in the compiled description as an object in the target language. The slots of the object include:

- A pointer back to the AST node containing the cache.
- A *valid* flag indicating whether the cache has been notified.
- A hash table containing the tuples of the relation.
- A list of *backlinks* pointing to DSL records that refer to this relation as their successor.

By using a hash table to store the tuples, we make sure that each tuple is represented only once. It is possible for a relation generator to yield the same tuple multiple times. While this cannot affect the semantics, it can have an adverse affect on efficiency. Maintained relations thus serve the subsidiary purpose of duplicate suppression.

Queries to a maintained relation are handled by the `GEN` method of the cache, which simply enumerates the contents of the hash table. When the cache is invalidated, it is reloaded using the `FILL` method, which implements the uncached semantics of the relation. As in the case of functional attributes, the `GEN` method is called indirectly via a wrapper function, which, in the case of a non-maintained relation, implements the uncached semantics directly.

A relation cache must be invalidated if its contents no longer agree with those that would be generated by its `FILL` method. In general, we can determine the results of the `FILL` method only by invoking it, but we only do so when a conservative approximation indicates that a change is possible. In exactly the same manner as for functional attributes, we propagate a status attribute for each relation (maintained or non-maintained) which is **true**, indicating a potential change if:

- The value of a scalar attribute (including object references) referenced in the relation body has changed.
- A selection of an object component via a reference-valued attribute has been notified, indicating that the value of the component has changed.
- The status attribute of a functional or relational attribute called within the relation body has the value **true** indicating that the relation may possibly no longer yield the same result.
- The identity of the node from which the status value for another functional or relational attribute was received has changed due to a modification to the structure of the tree.

```
method Block.VISIT_1(Ctx)
  ...
  % Invoke child visit to declaration list.
  % Result is status of attribute 'Decls.Binds'.
  Binds_STATUS = self.Decls.VISIT_1()
  % If the child pointer has changed, the status value from the
  % child visit is not reliable.
  Binds_STATUS = Binds_STATUS or self.CHILD_MODIFIED[1]
  % Retry maintained relation if needed, determining exact change status.
  Binds_STATUS = Retry_Cached_Tuples(self.BodyCtx.Binds, Binds_STATUS)
  when Binds_STATUS then
    % Notify dynamic successors.
    NLDL_Notify(self.NLDL_Binds)
  ...
  % Since this is the last visit to 'Block', clear the
  % 'CHILD_MODIFIED' flags.
  CHILD_MODIFIED[1] = false
  ...
  return
```

Figure 8.1: Evaluating (updating) a maintained relational attribute.

Changes to the tree structure are recorded in a vector of *child-modified* bits in each node, one for each child. The incremental parser sets a bit whenever it changes the value of the corresponding child pointer. These bits are cleared upon exit from the last visit to each node.[2]

In the case of a non-maintained relation, we simply accept this approximation as the authoritative change information. In the maintained case, the incoming status value is used to determine if the cache should be refilled. If a refill is required, the evaluator compares the new contents with the old, and indicates a **true** outgoing status only in the case that the contents of the relation have actually changed. Figure 8.1 shows the evaluation of the relational attribute `BodyCtx.Binds` in the `Block` statement. The actual cache update is performed by a runtime support procedure shown in Figure 8.2. Each tuple contains a flag bit `MARK`, which simplifies determination of whether the cache contents have changed following update.

When a relation cache is filled, dynamic dependencies may be created that are not covered by the incoming status value. These must be charged to the relation cache itself, so that notification via these dependencies will result in update of the cache. We thus extend the DDC once again, so that it may now be a relation cache, a mapping record, or a node/visit pair. Whenever a dynamic dependency link is created that indicates a relation cache as its successor, the link (whether an NLD or a DSL) must be added to the backlinks list belonging to the relation cache. Notification of a relation cache clears the `VALID` flag associated with the cache, and marks the node containing the cache for visits during the AST traversal. The cache will thus be considered for update if either it has been notified, or if it receives an incoming **true** status value. In the former case, the update is performed only if the `VALID` flag is false, as the node visit may have been scheduled on behalf of

[2] A single child-modified bit per node can be used by setting the bit in the new child rather than the old parent. This is not done in our present implementation for obscure technical and historical reasons specific to the incremental parser and AST representation that we use.

```
function Retry_Cached_Tuples(cache, status)
  if status == false and cache.VALID then
    % Cache cannot have changed.  Relation is up-to-date.
    return false
  else
    % Cache may have changed.  Regenerate cache contents.
    table = cache.HASHTABLE
    backlinks = cache.BACKLINKS
    cache.VALID = true
    cache.BACKLINKS = nil
    changed = false
    Retract_Support(backlinks)
    % The relation cache is the DDC for the generation of the relation.
    let *ddc-context* = cache in
      cache.FILL(lambda(a1, a2, ..., aN)
                  {
                     tuple = Hash_Lookup(table, [a1, a2, ..., aN])
                     if tuple == nil then
                       % New tuple.
                       changed = true
                       tuple.MARK = true
                       tuple.ARGS = [a1, a2, ..., aN]
                       Hash_Enter(table, [a1, a2, ..., aN])
                     else
                       % Old tuple.
                       tuple.MARK = true
                  } )
    Commit_Retractions(backlinks)
    foreach tuple in table do
      if tuple.MARK then
        % Clear 'MARK' in preparation for next update.
        tuple.MARK = false
      else
        % This pre-existing tuple is no longer a member of the relation.
        changed = true
        Hash_Remove(table, tuple.ARGS)
    return changed
```

Figure 8.2: Updating a maintained relation.

some other attribute. Figure 8.3 shows the changes required to the handling of DSLs, i.e., dynamic dependencies generated by cached function calls. Similar changes are needed for NLDs, i.e., dynamic dependencies generated by component selections.

8.4 Collections

Relational attributes must be propagated locally from node to node like any other attribute. ADL language descriptions typically provide their interface to clients via a number of global relations declared at top-level within the language description. Ordinary relational attributes require that the clauses that define them be included within the same scope as the declaration of the attribute. It is thus not possible to define the contents of an ordinary global relation using clauses appearing with the AST operators. This raises the question of how information computed during attribution of the AST can be transmitted to the exported global relations.

A collection is a special kind of relation that can be declared only at top-level, and whose defining clauses may appear anywhere within the language description. Collections are restricted in that they may not be queried within the language description, but only by external clients. As a result, collections cannot introduce any dependencies between attributes. The execution of the clauses of a collection may be interleaved in arbitrary order with other attribute evaluations. This obviates the primary reason for restricting conventional attributes to node-by-node propagation, namely, the need to provide a local description of the attribute dependencies from which a feasible global evaluation schedule may be derived.

In the analyzer for **Example**, a collection is used to associate error messages for the user with the AST nodes to which they apply:

```
collection Error(Node, String);
```

Within the AST operators, error messages are conditionally asserted into the collection by rules such as this one:

```
Error(Name, "Multiply-declared identifier") :-
    TypeDecl.Ctx.Duplicate(Name.Text, TypeObj);
```

The contents of the collection are the complete set of tuples for which the condition is satisfied. Rules asserting tuples unconditionally, or which assert multiple tuples are also permitted, just as in the case of an ordinary relational attribute.

8.5 Implementing Collections

Collections admit a particularly simple and efficient method of incremental evaluation. The `SUPPORTS` lists for each node contain backpointers to the collection tuples asserted on the corresponding visit. Upon entry to a visit procedure during an incremental update, the tuples asserted in the previous visit are removed, and are regenerated during the current one. Collection tuples are stored in a hash table to eliminate duplicates, since it is possible that the same tuple may be asserted more than once.[3] Each tuple thus contains a *multiplicity* count that indicates how many assertions are outstanding. Deletion of a tuple merely decrements its multiplicity, and the tuple is

[3] Collection tuples may actually be stored in multiple hash tables, as many as one per column of the relation, permitting efficient indexed query. This feature has not been fully implemented at the ADL source-language level, however. By default, every collection is indexed on its first column.

```
function Add_Dynamic_Predecessor_Mapping(mapping)
  context = *ddc-context*
  dsl = Create_DSL()
  dsl.SUCCESSOR = context
  dsl.MAPPING = mapping
  DSLL_Insert(dsl, mapping.SUCCESSORS)
  % Register backlink, depending on kind of DDC in effect.
  typecase context of
    operator:
      visit = *ddc-visit*
      context.SUPPORTS[visit] = cons(mapping, context.SUPPORTS[visit])
    mapping:
      context.BACKLINKS = cons(mapping, context.BACKLINKS)
    relcache:
      % DDC is a relation cache.
      context.BACKLINKS = cons(mapping, context.BACKLINKS)
```

```
function Notify_Mapping_Successors(mapping)
  foreach succ in mapping.SUCCESSORS do
    typecase succ of
      operator:
        Notify_Change(succ)
      mapping:
        cache = succ.CACHE
        cache.RETRIES = cons(succ, cache.RETRIES)
        Notify_Change(cache.ASTNODE)
      relcache:
        % DDC is a relation cache.
        % Mark cache invalid and force a visit to its AST node.
        succ.VALID = false
        Notify_Change(succ.ASTNODE)
```

Figure 8.3: Revisions for DSL handling in the presence of maintained relations.

actually removed from the hash table only when its multiplicity drops to zero. To avoid extraneous removal and re-creation of tuples, we refine this scheme further, pushing tuples whose multiplicity has dropped to zero onto a temporary stack, which is then scanned at the end of the update. If the tuple has been re-asserted, raising its multiplicity to a non-zero value, the tuple is retained. Tuples whose multiplicity remains zero are removed. With this refinement, the algorithm can provide notification to external clients of exactly which tuples were added to and removed from a collection during an incremental update. We provide a means for clients to register interest in these notifications via callback procedures.

8.6 Related Work

Abramson's Definite Clause Translation Grammars [1] are essentially attribute grammars in which all attributes are synthesized relational attributes. His implementation is embedded in Prolog, whose "call by unification" parameter passing scheme allows data to flow in both directions, thus obtaining the effect of inherited as well as synthesized (non-relational) attributes. He does not discuss incremental evaluation.

Sataluri and Fleck [69] defines a variant of attribute grammars in which both inherited and synthesized relational attributes are permitted. They are concerned principally with extending the expressiveness of attribute grammars as a specification formalism, and do not discuss implementation other than to note that the logic program induced by a particular AST may be evaluated by conventional means as an ordinary logic program.

Our notion of collections is borrowed from Horwitz [39] and Horwitz and Teitelbaum [38], who carry the idea further by allowing the definition of incrementally-maintained derived relations. Although they call their technique "relational attribution," the relations are all defined at top-level, and do not decorate the tree. Only the assertions actually appear within the AST operators.

Colander [4] provided collections as first-class objects that could be transmitted as data values within the language description. We eschew this generality in the interest of efficiency. Our relational attributes are much more amenable to static analysis than Colander's first-class collections.

Hedin's objects [32] may possess a special kind of component called a "collection," distinct from our usage of the term, which may be constrained to contain certain members remotely at sites where a reference to the object is available. We considered generalizing our collections to permit collections local to an AST operator, including use as an object component, and to permit other attributes to depend on their contents. Because information flows into a collection in the reverse direction from the nominal direction of attribute flow, however, the usual static analysis would fail. Unlike Hedin, we were not willing to rely on unchecked manual analysis. For this reason, we rejected the extension. It came to our attention recently, however, that the fibering mechanism developed in the next chapter provides a simple solution to this difficulty, suggesting that our collections might be extended to encompass Hedin's without compromising the integrity of the automated dependency analysis.

Chapter 9

Incremental Evaluation in Review

In the last three chapters, we have introduced a great deal of complicated machinery to support objects, functional attributes, and relational attributes. Underlying these mechanisms, however, are a small number of simple ideas. In this chapter, we attempt a unifying overview, and then discuss performance issues and potential improvements.

9.1 A Unified View of Incremental Evaluation

We begin by observing that the need for all this machinery arises solely due to our concern with incremental evaluation. In a batch evaluator, the implementation of objects, functional attributes, and relational attributes is a straightforward exercise. Our approach to incrementality has two essential facets that, in combination, account for the proliferation of mechanism we have seen. The first facet is the pervasive use of caching, which appears in several guises. Though different mechanisms are used to cache visits, functions, and relations, the essential effect is the same in each case. The cached computation may be avoided if we know that it must return the same value as it did on a previous evaluation when the cached result was determined. The second facet is the refinement of coarse static dependencies with precise dynamic dependencies. Maintaining precise dynamic dependency information at runtime is essential to exploit the potential for incremental evaluation in the presence of aggregate attributes. By assuring that all such dependencies are covered by those assumed statically at evaluator generation time, however, we avoid the need to perform scheduling at runtime. Dynamic dependencies are used only for cache invalidation, which simplifies their management considerably.

9.1.1 Caching

Maintaining caches for functions and relations may have some value even in a batch-mode evaluator. Caching function calls may significantly improve the efficiency of some functions that are repeatedly called at the same argument values. For example, the doubly-recursive rendering of the Fibonacci function requires exponential time when naively implemented, but only linear time when memoized. Caching of relations in a generator-based implementation such as ours may save time when the same relation is queried many times. Nonetheless, the essential role of caching in our system is to permit incremental evaluation, toward which all of the particulars of our design are directed. Other functions that may be served by caching are incidental to our design goals and to this discussion. Caching, including that of visits by retention of attributes at `anchor` points in the tree, is the enabling mechanism for incremental evaluation. Only by retaining the result of a computation in an earlier analysis is it possible to avoid performing that computation again in a subsequent analysis. Once a mechanism for retaining and reusing results is in place, however, it is necessary to avoid re-using cached results that are no longer valid, lest correctness be compromised.

We use two distinct strategies to avoid using cached results that are no longer valid. When the predecessors of the cached computation are immediately available and cheaply compared, e.g., the inherited scalar attributes of a cached visit, we can simply compare the current values with the values saved from the previous evaluation, on the basis of which the cached result was computed. This treatment is not appropriate for long-distance attribute dependencies, however. In that case, it most appropriate to test for a change at the time that an attribute is updated, and then invalidate the affected caches remotely via notification. Although the mechanics of notification for visits, function caches, and relation caches all differ in detail, the effect is always to cause the re-evaluation of (at least) the innermost cached successors of the changed attribute. All uncached successors will necessarily be re-evaluated up to and including the innermost cached successors, as it is only a cache "hit" that can cut short a full batch-mode re-evaluation. The dynamic dependency context (DDC) represents the innermost cached successor of all function calls and selections performed while it is in force, thus allowing the successor to be identified correctly by the dynamic dependency links. For functions and relations, change detection itself is intimately tied up with caching, as it is only within a function or relation cache that the extension of a function or a relation is explicitly represented and thus subject to examination and comparison. To allow for the possibility that a function or relation might not be provided with a cache, we may conservatively approximate the change status of the function or relation by assuming that it must have changed if any of its predecessors have changed. The effect is to fall back temporarily to batch-mode operation when caches are not available, as their lack precludes both the possibility of cache hits and the availability of precise change information.

The notification of a selection when an object component has changed is actually a form of cache invalidation. When the target of the notification is a tree node, the visits that are forced due to marking of the spine of the AST include all of those that might compute a synthesized attribute value invalidated by the change. If the target is a function cache (i.e., a cached mapping) or a relation cache, the spine is also marked, but with the different purpose of merely assuring that the evaluation traversal reaches the affected cache, which is itself marked invalid.

9.1.2 Dynamic Refinement of Static Dependencies

In the classical dependency model, aggregation of values into composite data structures precludes the ability to account for their dependencies separately. The dependency of another attribute upon a component of a composite data structure is carried by the dependency on the structured value itself, which is then in turn dependent upon its components. The result is that the dependencies involving the components are coalesced, yielding a coarse approximation to the true dependencies. In some cases, it would be possible to rewrite the AG so that the components were separate attributes, or to use techniques such as fibering (presented in the following chapter) to avoid this loss of precision. In general, however, e.g., when the number of components is determined at runtime, or is potentially unbounded, we cannot expect to obtain precise information statically at evaluator generation time.

Our strategy, then, is to perform an initial static analysis based on the classical imprecise (aggregated) dependencies, and to schedule the attribute evaluations and construct the visit sequences based on that analysis. Whether a scheduled evaluation will actually be performed at runtime depends on where caching takes place and whether cache retrievals hit or miss. Since we maintain a separate dynamic dependency link at runtime for each object component, the invalidations performed when a component is changed affect only the cached visits (or other successors) that depend on that component via selection. For the case of function caching, we similarly "split" the original coarse dependencies on the entire function value into separate dependency traces for the value of the function at each distinct tuple of arguments. The more precise dynamic dependency links result in accurate cache invalidation, without requiring dynamic scheduling.

9.2 Using Caching Effectively

The maintenance of caches and dynamic dependency links represents a substantial cost in space and time. In general, it is a good idea to cache as little as possible consistent with maintaining a sufficiently small granularity of incremental re-evaluation. For example, a reasonable policy might be to re-analyze at the granularity of individual top-level forms, as is common in hand-coded incremental analyzers. In our description for **Example** in Chapter 4, we re-analyze at the level of individual declarations and statements.

To a first approximation, the granularity of re-evaluation is controlled by the use of the `anchor` pragma, which relates to the granularity in a fairly intuitive way. Failure to provide caches for functions and relations at appropriate points, however, may result in spurious notifications that thwart the description writer's intent as expressed in these pragmas. In the case of functional and relational attributes whose values may be accessed non-locally, i.e., which appear as object components, the effect of spurious notifications due to imprecise change information may be widespread. Caches should thus almost always be maintained for such attributes. Caching assures that the outgoing change status computed for the function or relation is absolutely precise. In other cases, where the attribute is accessed only locally, the benefits of caching are more limited, while nonetheless incurring the full cost. The cost of caching includes not only the time involved in manipulating the cache entries, but the space overhead for the dynamic dependency links. Retrying a large relation cache, however, may be expensive, and if caching a local predecessor can reduce the number of retries, it may be worthwhile. There are no general rules, and careful consideration of the details of the particular language description is in order. Since the caching pragmas can affect neither the correctness nor the well-formedness of the description with respect to circularity, the caching strategy can be tuned based on experimental feedback.

9.3 Improving the Treatment of Relations

Our present treatment of relational attributes does not provide the same degree of fine-grained incrementality that we provide for functions. It may thus be advisable in some cases to avoid their use where they would otherwise be highly appropriate. In this section, we discuss avenues for possible improvements in future work.

Our implementation of relations does not exploit the potential for a dependency successor of a relation to depend only upon the presence or absence of a specific tuple or subset of tuples. Given our operational semantics, however, in which the instantiated arguments of a query are not used to restrict the set of tuples enumerated by the generator function for the queried relation, tracking dependencies in this way doesn't make sense. Every query must necessarily depend on any change to the relation, since the query gives the relation no information about which tuples it actually cares about.

In Prolog, a relation can be defined with no prior commitment to the information that must be supplied by its queries. The implementation of Prolog relies, however, on the use of logical variables and unification, which we have avoided by design in ADL. In ADL, every relation argument is an output argument, thus the queried relation receives no information from the query context.

Input arguments, however, could be handled as well, without the introduction of logical variables, if the relation could be compiled with the knowledge that every query would provide values for those arguments. The input arguments would simply become arguments to the generator function implementing the relation, rather than to the continuation that the generator invokes. We thus propose that relations be declared with directional annotations on their arguments, specifying which are required to be expressions, and thus evaluable (input arguments), and which may be pattern variables, and thus bound by the query itself (output arguments). For example, the `Binds` relation in the example of Chapter 4 could be declared as follows, where the "-" symbol indicates an input argument and the "+" symbol indicates an output argument:

```
relation Binds(-String, +Entity);
```

Collectively, the annotations define the *mode* of the relation, in the same manner as the optional mode declarations supported by many Prolog compilers. Relations may then be indexed on their input arguments, allowing faster queries, as well as enabling fined-grained tuple dependencies.

While we suspect that usage at a single mode will be sufficient for most relations, mode restrictions do imply a loss of expressive power. A relation in which all arguments are of output mode is the most general in the sense that it can be queried at any mode, though sometimes rather inefficiently. Relations that restrict some arguments to be instantiated, however, are essentially set-valued functions. Because a mode restriction is essentially a pragmatic issue, we would like to treat mode annotations as pragmas, to be concealed in the body of the analysis description. Unfortunately, mode annotations would have to be included in the schema as well, as they would be a necessary part of the interface to an exported relation as seen by its clients.

Statically moded relations restricted to require instantiated input arguments may be implemented in a manner that extends the existing treatment of maintained functions in a relatively straightforward way. Queries may record dependencies using a list of dynamic successor links for each relation tuple. Relation queries differ from function calls, however, in that the query may "return" more than once, yielding multiple result tuples, and may also fail entirely, yielding no tuples at all. In the event that the query fails, a dependency on this fact must be recorded, as the subsequent addition of a new tuple may allow it to succeed. Ballance's Colander system uses a tuple placeholder called a *hole*, which is a partially-instantiated tuple (containing variables in general) that matches only those ground tuples that satisfy the query that created it. When a new tuple is added, any holes that match it are deleted, and their dependency successors are notified of the change. In a statically-moded context such as ours, we can index a relation on all of its input arguments in the same way that we index a function cache on the arguments of the function. A check for matching holes, then, becomes as efficient as an ordinary tuple lookup.

The method we suggest here further extends the "notify and retry" paradigm that is pervasive in our approach to incrementality. Horwitz [39] implemented a scheme that adheres more closely to the traditional "change propagation" model, in which sets of tuples added and deleted from a relation are propagated to its successors in an apparently eager, data-driven manner. In reality, the implementation of her method uses generator functions much like ours. Although Horwitz's relations do not decorate the tree as attributes, her generators could most likely be adapted to support such attribution in the same manner as ours. One advantage of her method is that it can handle queries at any mode, and, like ours, need not maintain every relation in materialized form. On the downside, however, it is considerably more complex. Additionally, separate provision must be made for non-local dependencies, while the method we propose handles both fined-grained dependencies and non-local dependencies with a common mechanism. A hybrid approach might be workable, and would be worth investigating.

9.4 Performance Measurements

We have implemented the incremental evaluation method described here as an extension to the Pan language-based environment. The particulars of the implementation are presented in Chapter 11. In this section, we report on the performance of the method as embodied in this prototype implementation. We have instrumented the implementation to measure the actual execution time during semantic analysis, as well as to maintain counts of interesting evaluation-time events, including node visits, cache hits and misses, and notifications. Both the generated code and the runtime support routines are written in Common Lisp, and favor clarity and implementation expediency over execution speed, with little attempt at optimization. For this reason, the absolute timings should not be taken as indicative of the performance to be expected of a production-quality implementation. On the other hand, the execution times for incremental updates relative to that for the initial analysis provides a measure of the performance increase due to incrementality. The event counts provide insight into the operation of the algorithm, as well as some evidence for where the costs lie.

For our experiments, we used the language description presented in Chapter 4 for the language **Example**. We performed the measurements using the program in Figure 9.1. For compactness, we have elided three repeating groups of statements, as indicated in the figure. The actual program is 613 lines long. Selected lines are labeled with a letter of the alphabet, and will be referred to by that label in subsequent discussion. Where a labeled line appears within a repeating group of statements, the label applies to the earliest occurrence.

In the first set of experiments, each edit consisted of adding or removing a single line, re-analyzing after each change. The table in Figure 9.2 shows the results. The first line of the table gives the results for the initial full analysis of the program before any changes were made. The program was loaded, analyzed, then loaded and analyzed again so that the timings for the initial analysis would not be penalized for the initial page-in of the analyzer code. A sequence of changes were then made, indicated using the notation *l*+ for a line insertion and *l*- for a line deletion, where *l* is the alphabetic label of the affected line. In each case, we report the number of errors discovered by the analysis, the number of node visits actually performed by the evaluator, the number of visits that were pruned by visit cache hits, and execution times for two separate runs. Execution times are reported in milliseconds, as reported by the Common Lisp function `get-internal-runtime`. While this function returns the time in units of milliseconds, the elapsed times appear to be quantized to multiples of approximately 1/60th of a second, or 16.67 milliseconds, and should be interpreted accordingly. In a couple of cases, we repeated a measurement that had obviously been affected by garbage collection or unusual system activity.

We find that the algorithm is quite effective in pruning node visits, although the effect is noticeably more pronounced, as we might expect, for statements as well as for declarations of names that are not widely used. As currently coded, our language description requires that every declaration in a contour be visited whenever any of its sibling declarations is inserted or deleted, due to the way that we test for multiply-declared identifiers.[1] When interpreting the visit counts, the reader should remember that a visit cache hit prunes the traversal of an entire subtree, which may contain an arbitrary number of nodes. A better measure of the work saved by visit caching is the difference between the number of visits performed during the incremental analysis and the number performed in the initial analysis.

Figure 9.3 shows detailed event counts for the same sequence of edits. They show the cost of our current naive treatment of relations. Every time a declaration is added or removed, the `Binds`

[1] A more efficient approach would delegate duplicate detection entirely to the binding contour object. Declarations would then have to include a node reference for the declared name in the `Binds` relation, so that errors could be asserted at the proper location.

```
declare
  var r : integer;                       (A)
  type aryptr = pointer to ary;          (B)
  type ary = array [ 5 ] of integer;     (C)
  var v : ary;                           (D)
  var w : aryptr;                        (E)
  var x : array [ 10 ] of aryptr;        (F)
  var y : integer;                       (G)
  var z : array [ 50 ] of integer        (H)
begin
  declare                                Repeat this block 25 times
    type p1 = pointer to p2;             (I)
    type p2 = pointer to ary;            (J)
    var s : p1;                          (K)
    var t : p2;                          (L)
    var u : integer;                     (M)
    var v : array [ 5 ] of integer       (N)
  begin
    y := 1;
    t^[y] := u;
    y := u + z[5];                       (O)
    w^[1] := u;
    u := v[u];
    declare
      var v : integer                    (P)
    begin
      v := v + u                         (Q)
    end;
    v := t^;
    u := u + u;
    t := s^
  end;
  ...
  y := 1;                                Repeat next 5 statements 5 times
  z[5] := y;
  y := y + z[y];                         (R)
  w^[1] := y;
  v := w^;
  ...
  r := r + 1;                            (S)
  y := 1;                                Repeat next 5 statements 5 times
  z[5] := y;
  y := y + z[y];
  w^[1] := y;
  v := w^;
  ...
  y := y + 1
end
```

Figure 9.1: Example program for measurements of incremental parser performance.

Change	Errors	Timings			Avg. Time	Visits	VC Hits
IN	0	2100	2033	2067	2067	4651	0
A−	4	33	50	50	44	88	9
A+	0	50	50	50	50	99	9
B−	2	250	233	216	233	709	174
B+	0	250	283	250	261	716	174
C−	27	666	634	683	661	1730	273
C+	0	716	717	700	711	1640	248
D−	20	100	83	100	94	171	30
D+	0	67	117	100	94	181	29
E−	90	267	300	284	284	705	174
E+	0	316	300	300	305	716	174
F−	0	33	33	33	33	74	5
F+	0	34	50	50	45	87	5
G−	274	583	583	600	589	1282	138
G+	0	600	633	567	600	1293	138
H−	91	384	334	316	345	797	172
H+	0	350	350	350	350	801	172
I−	1	50	33	50	44	78	14
I+	0	33	34	17	28	88	14
J−	2	50	50	50	50	98	14
J+	0	50	34	50	45	107	16
K−	2	33	50	50	44	77	14
K+	0	50	50	50	50	88	14
L−	6	50	50	34	45	96	16
L+	0	50	50	50	50	107	16
M−	18	67	83	100	83	133	17
M+	0	83	100	83	89	144	17
N−	0	33	50	50	44	85	15
N+	0	50	33	50	44	98	15
O−	0	0	17	34	17	30	14
O+	0	17	17	17	17	41	14
P−	2	16	34	17	22	41	16
P+	2	0	16	16	11	46	16
Q−	0	17	0	16	11	31	19
Q+	0	17	0	17	11	38	19
R−	0	17	17	16	17	24	11
R+	0	16	0	0	5	35	10
S−	0	0	0	17	6	22	13
S+	0	16	0	0	5	29	12

Figure 9.2: Timings and visit statistics for line insertion and deletion.

relation for its contour must be reconstructed, even though all but one tuple was previously contained in the relation. Because relations do not track the successors of individual tuples dynamically, it is then necessary to regenerate the cache for `VisibleBinding` as well, since `Binds` is in its closure.

We observe that a large number of non-local dependency links are constructed for dynamic selections. This isn't surprising, since every call to `BaseType` or `EquivTypes` registers at least one non-local dependency, and one or both of these functions is called at least once in every statement. Nonetheless, the fact that the number of such registrations is a significant fraction of the total number of attributes evaluated is a cause for concern.

In a second set of experiments, we made single-token changes within a number of declarations. The results are shown in Figure 9.4.

	EV	FH	FM	MN	FV	FR	FB	MV	MC	RV	RB	TN	TO	ND	NS	NT
IN	6691	664	358	0	0	0	51	0	0	0	51	183	0	1759	183	0
A−	91	9	0	2	0	0	1	7	1	0	1	0	7	6	7	11
A+	99	10	0	2	0	0	1	7	1	0	1	1	7	10	8	7
B−	1244	97	0	2	25	0	1	7	1	25	1	0	7	192	7	99
B+	1252	98	0	2	25	0	1	7	1	25	1	1	7	286	8	52
C−	2868	322	0	52	0	25	1	7	26	25	1	0	7	728	32	320
C+	2852	323	0	52	0	25	1	7	26	25	1	1	7	861	33	188
D−	254	27	0	10	0	0	1	7	1	0	1	0	7	45	7	27
D+	262	28	0	10	0	0	1	7	1	0	1	1	7	76	8	7
E−	1244	122	0	70	0	25	1	7	26	25	1	0	7	105	7	97
E+	1252	123	0	70	0	25	1	7	26	25	1	1	7	286	8	7
F−	62	7	0	0	0	0	1	8	0	0	1	0	7	5	7	7
F+	72	8	0	0	0	0	1	8	0	0	1	1	7	6	8	7
G−	2444	299	0	162	0	25	1	7	26	25	1	0	7	336	7	281
G+	2452	300	0	162	0	25	1	7	26	25	1	1	7	610	8	7
H−	1440	167	0	70	0	25	1	7	26	25	1	0	7	186	7	97
H+	1447	168	0	70	0	25	1	7	26	25	1	1	7	276	8	7
I−	83	12	1	1	1	0	1	10	1	1	1	0	5	8	5	8
I+	92	13	0	1	1	0	1	10	1	1	1	1	5	13	6	6
J−	123	16	1	2	1	0	1	9	1	1	1	0	5	17	5	16
J+	131	17	1	2	1	0	1	9	1	1	1	1	5	27	6	10
K−	84	12	1	1	1	0	1	9	1	1	1	0	5	8	5	7
K+	92	12	1	1	1	0	1	9	1	1	1	1	5	13	6	5
L−	123	17	1	3	1	0	1	10	1	1	1	0	5	15	5	11
L+	131	18	0	3	1	0	1	10	1	1	1	1	5	27	6	5
M−	199	28	1	10	1	1	1	10	2	2	1	0	5	26	5	23
M+	207	29	0	10	1	1	1	10	2	2	1	1	5	44	6	5
N−	104	16	0	2	1	0	1	10	1	1	1	0	5	20	5	9
N+	114	16	0	2	1	0	1	10	1	1	1	1	5	18	6	5
O−	48	2	0	0	2	0	0	0	0	2	0	0	0	6	0	0
O+	72	5	1	0	2	0	0	0	0	2	0	0	0	12	0	0
P−	56	5	0	2	2	0	1	1	1	2	1	0	0	6	0	4
P+	62	5	1	2	2	0	1	1	1	2	1	1	0	6	1	0
Q−	40	0	0	0	3	0	0	0	0	3	0	0	0	0	0	0
Q+	56	2	2	0	3	0	0	0	0	3	0	0	0	6	0	0
R−	42	2	0	0	1	0	0	0	0	1	0	0	0	6	0	0
R+	67	6	0	0	1	0	0	0	0	1	0	0	0	14	0	0
S−	34	1	0	0	1	0	0	0	0	1	0	0	0	2	0	0
S+	47	2	1	0	1	0	0	0	0	1	0	0	0	6	0	0

EV	Scalar attribute evaluations
FH	Function cache hits
FM	Function cache misses
MN	Mappings requiring notification of successors
FV	Function caches visited and found valid (no retries needed)
FR	Function caches requiring mapping retries
FB	Function caches requiring full reconstruction
MV	Function mappings found to be valid upon retry or reconstruction
MC	Function mappings that changed upon retry or reconstruction
RV	Relation caches visited and found valid
RB	Relation caches found to be invalid and reconstructed
TN	New tuples added during relation cache reconstruction
TO	Old tuples still valid after relation cache reconstruction
ND	NLD links created for dynamic selections
NS	NLD links created for static selections
NT	NLDL successor notifications performed

Figure 9.3: Detailed event counts for line insertions and deletions.

Line	Change	Errors	Timings			Avg. Time	Visits	VC Hits
C	`5` $\rightarrow$ `10`	25	467	483	467	472	1132	230
C	(restore)	0	500	500	466	489	1132	230
F	`10` $\rightarrow$ `20`	0	0	0	0	0	30	13
F	(restore)	0	17	17	33	22	30	13
H	`50` $\rightarrow$ `100`	0	267	267	267	267	748	181
H	(restore)	0	266	250	267	261	748	181
B	`ary` $\rightarrow$ `integer`	45	217	250	233	233	659	182
B	(restore)	0	233	250	250	244	659	182
C	`integer` $\rightarrow$ `ary`	61	466	517	517	500	1132	230
C	(restore)	0	483	483	483	483	1132	230
F	`aryptr` $\rightarrow$ `integer`	0	0	17	17	11	30	13
F	(restore)	0	33	0	0	11	30	13
J	`ary` $\rightarrow$ `integer`	2	17	17	17	17	62	18
J	(restore)	0	33	34	34	34	62	18

Figure 9.4: Timings and visit statistics for intra-line changes.

Chapter 10

Fibering

One of the strengths of attribute grammar technology is the use of static analysis to prove strong properties of the language description independent of any particular program to be analyzed. In particular, the circularity test rejects descriptions that may give rise to ill-founded attribute definitions. Unfortunately, some common language features are most naturally defined in a form that produces a circular AG, requiring tedious circumlocutions to render the description in a noncircular form. In this chapter, we present a technique that allows certain circularities to be admitted without compromising static verifiability or statically scheduled evaluation.

10.1 The Problem

Many programming languages permit the definition of new composite data types in terms of existing ones, e.g., arrays and records. Such types may be given names and subsequently referenced in the same manner as the built-in primitive types. Data types may be recursive, containing (pointers to) one or more instances of the type itself as subcomponents. Within a static semantic analyzer, composite types are most conveniently represented as linked data structures in which the nodes model the type constructors and the links model the component containment relationships. Recursive types thus give rise to cyclic (re-entrant) type descriptors at analysis time. Unfortunately, cyclic data structures cannot be created within a conventional attribute grammar, due to the use of an applicative-order ("call by value") evaluation mechanism in the underlying functional programming language.

It is possible to model a cyclic graph, such as the descriptor for a recursive type, without actually building a cyclic data structure. We might, for example, represent the edges of the graph in a separately maintained relation, allowing all of the type descriptor records, i.e., the nodes of the graph, to be created prior to asserting anything about their relationships. Since the edges of the graph are directed in this case, a functional mapping suffices. A clever trick, used by Watt in a published attribute grammar [80] for Pascal, exploits the fact that any cycle in a type descriptor must contain at least one reference to a named type. In Pascal, every such cycle is completed by a named type reference occurring in the definition of a pointer type. Watt's grammar simply leaves such references in symbolic form, and performs an identifier lookup each time the referent is required during subsequent usage of the descriptor. To avoid unintended interference from other declarations using the same name within different scopes, a unique new name is invented for use within the type descriptor, replacing the one supplied by the user. Although an adequate solution in some cases, neither of these methods is as convenient or as efficient as a directly linked re-entrant representation.

Another related problem arises in languages that do not restrict the textual order in which declarations are written, i.e., that do not enforce the requirement that declarations of names precede their usage within other declarations. Consider the language **Example** and the analyzer that we defined for it in Chapter 4. In that language, both named types and variables may be declared. A variable may be declared of a type defined in a subsequent declaration. Named types may be defined in terms of other types, but the use of a variable name within a type declaration is an error. While we may deduce from an examination of the attribute grammar that the descriptor for a named type will never contain a reference to that of a variable, this fact will not be apparent to the classical circularity test, since both kinds of descriptors are aggregated within the binding environment. Thus the classical dependency analysis will determine, incorrectly, that the descriptor for a variable, which depends on the binding environment (to resolve type names appearing within its declared type) may depend on the variable itself, which is also bound in the environment.

In a language requiring declaration before use, the binding environment may be threaded through a sequence of declarations such that each declaration is analyzed in the context of the bindings established by the preceding declarations only. This strategy makes it immediately apparent to the dependency analysis that mutually dependent or self-dependent declarations are precluded. Threading of an attribute through a sequence, however, is undesirable in an incremental context, as discussed in Chapter 5. A better strategy is to process declaration sequences initially as if there were no restriction on declaration order. An additional test is then applied to the relative textual locations of each identifier usage and its corresponding declaration. In addition to improved incremental efficiency, this approach provides more meaningful error diagnostics, as illegal forward references are identified as such rather than reported as undefined names.

Languages without restrictions on declaration order, such as **Example**, can be accommodated in a classical attribute grammar by threading the environment through each declaration sequence twice. On the first pass through, only the bindings for type names are entered into the environment. On the second pass, type names are resolved in the environment constructed in the previous pass, while building a new and complete environment containing both the type-name bindings established in the previous pass and new bindings for the variable names. At runtime, the generated attribute evaluator processes the declarations in a pair of sequential passes mirroring the threading of the environment. Thus, by coding the analysis description in this fashion, the user is essentially performing the evaluation scheduling by hand. It would be preferable to improve the generation-time analysis so that it could produce a comparable evaluation schedule from a more straightforward attribute grammar.

10.2 Overview of Fibering

In a static semantic analyzer written by hand in a conventional imperative language, circular type descriptors pose no problem. Within the declaration sequence constituting a single binding contour, the analyzer need not actually traverse any of the links within any type descriptor in order to build the descriptors or to resolve references to named types. After the type descriptors for the entire binding contour are complete, the type descriptors may be used in subsequent processing without restriction. The usual strategy, then, is to leave the slots containing pointers to the referents of forward-referenced names temporarily undefined until all of the declarations have been processed. At that time, all of the names will have been associated with (possibly incomplete) descriptors, and the analyzer may then backpatch the slots previously left undefined. The technique is correct provided that no undefined slots are accessed. This is usually easy to ensure by requiring that all of the slots are initialized before any of them are accessed.

In the underlying implementation, our attribute grammars are actually rendered in an imperative language, and attribute values are computed and stored sequentially, subject to the constraint that an attribute not be accessed before it is defined. The components of our objects are simply attributes themselves. A reference to an object can be created and transmitted elsewhere in the abstract syntax tree without first initializing the object's component attributes. It is simply necessary to ensure that each component is evaluated in every case before it is accessed, including remote access via an object reference. In the classical analysis method, this condition is assured by a conservative approximation that assumes that the object reference depends on all of the components. In the underlying imperative execution model, however, there is no essential requirement (as there is in the functional model) that the components be evaluated first.

In this chapter, we develop a mostly automatic and mechanically verified method for relaxing the conservative but restrictive approximation made by the classical analysis, more closely capturing the actual dependencies obtaining between selection instances and the specific components that they access. In doing so, not only do we permit the construction of cyclic data structures, but we also solve the second problem of the previous section, the conflation of dependencies involving distinct object types when the objects are transmitted via a common attribute. We call the technique *fibering*, as it has similar goals and some conceptual similarity to a method of the same name described in an unpublished paper by Farrow [22]. Fibering is a compile-time transformation that does not impact our runtime machinery in any way. The fibering algorithm will be described in detail in the following section. In summary, it proceeds as follows:

1. Determine the sets of attribute occurrences whose instances may be involved in a common dependency cycle, based on an initial approximation of the attribute dependencies in which each object is assumed to depend on all of its components.

2. For each attribute occurrence appearing within a cycle, determine which component occurrences of objects within the same cycle, if any, may have instances at runtime that could be accessed during the evaluation of the attribute occurrence. Such attribute occurrences are called *selection sites* for the component accessed. This analysis is a straightforward dataflow problem.

3. For any object included in a cycle, remove the dependency of the object on any of its components that are also members of the cycle. It is this step that actually breaks the cycles detected by the classical dependency analysis.[1] Each component from which dependencies are removed by the fibering algorithm is said to have been *fibered*.

4. Add control attributes and dependencies linking the fibered components within a cycle as predecessors to any of their corresponding selection sites that appear within the same cycle. These dependencies guarantee the soundness of access to the fibered components during the evaluation of the attribute occurrences within the cycles to which the components belong. Additionally, add control attributes and dependencies linking all fibered components of a cycle as predecessors to any attribute occurrence that depends on an attribute within the cycle, but does not share any common cycle with the component. These dependencies guarantee the soundness of access to the fibered components during the evaluation of attribute occurrences outside of the cycles to which the components belong.

5. Test the resulting attribute grammar for circularity. Fibering succeeds if the fibered AG is noncircular.

[1] As noted earlier in Chapter 4, only components declared as `delayed` are considered here. In this chapter, we will initially ignore this refinement, re-introducing it in Section 10.4.

Control attributes are attributes that have no value and for which no storage is allocated, but simply serve to express ordering constraints between imperative computations associated with their "evaluation." By appropriate usage of control attributes, operations with side-effects may be included in the attribute grammar with the assurance that they will be performed prior to operations that might observe them.[2]

10.3 The Fibering Algorithm

We implement fibering as an extension of our static evaluator generator for ordered attribute grammars (OAGs). Consequently, the resulting fibered AG must not only be noncircular, but must also belong the the OAG class. Furthermore, we use a conservative approximation to the true noncircularity condition when determining whether fibering is required. The approximate test can report a spurious circularity only in the case that the description is not in the OAG class, which in any event precludes the construction of an evaluator for the unfibered AG.

We will illustrate the operation of the fibering algorithm using the language description in Figures 10.1 and 10.2. This highly contrived example is intended to highlight the interesting aspects of the operation of the algorithm while remaining readable.[3] "Programs" in this language consist of strings of the form "`foo bar1 baz`," "`foo bar2 bar1 baz`," "`foo bar2 bar2 ... bar1 baz`" and so on. The relevant features of the analysis from the standpoint of fibering are that (1) a cyclic object of class `Cycle` is created, linked via its `Ref` component, and (2) the component `Int2` of the object depends on the value of an instance of the component `Int1` that is accessed indirectly via an object reference.

10.3.1 Locating Potential Cycles

The first step of the OAG evaluator construction algorithm computes an approximation to the dependencies among the attributes of each phylum, represented as a directed graph called a *summary dependency graph*. (Reps and Teitelbaum [67] use the term *TDS graph*, for "transitive dependencies of a symbol.") This approximation is conservative in that it covers the dependencies that exist at every instance of the phylum in every possible AST. Figure 10.3 shows the dependencies discovered among the attributes of the phylum `Bar`. The dependencies shown with a dotted line are implied by the others, as the summary dependency graphs are always closed under transitivity.

The local dependency graphs for each AST operator are then composed with the summary dependency graphs applicable at the parent and child interfaces of the operator. The summary dependency graphs serve to characterize the transitive dependencies induced by all possible contexts in which the operator may occur within an AST. If the resulting *augmented dependency graph* for each AST operator is noncircular, then the AG belongs to the *doubly noncircular* (DNC) [17] class of attribute grammars, and the construction proceeds to the next step. If a cycle is found, however, the classical OAG construction fails, as the OAGs are a subclass of the DNC AGs.

Our fibered attribute evaluator generator proceeds in the conventional way should the DNC test succeed, and attempts to construct a classical OAG evaluator. Should the test fail, however, it applies the fibering transformation and then attempts the classical OAG construction again on

[2] The notion of "dummy" attributes arises frequently in the literature, however the LIGA system [48, 78] is notable for its systematic use of such attributes to integrate imperative programming into attribute grammars in a reasonably clean manner.

[3] As it turns out, the **Example** language is a degenerate case for fibering, and is thus not suitable for this purpose. The description for Modula-2 presented in Chapter 12, on the other hand, is so large and makes such extensive use of fibering that it is completely intractable as a pedagogical example.

```
language body Fibers : Fibers is

  class Cycle
  requiring
    attribute Ref : Cycle
        delayed;
    attribute Int1 : Integer;
    attribute Int2 : Integer
        delayed;
  end Cycle;

  object NullCycle : Cycle
  where
    Ref = NullCycle;
    Int1 = 0;
    Int2 = 0;
  end NullCycle;

  phylum Foo;

  phylum Bar
  with
    context   CompIn : Cycle;
    attribute ObjOut : Cycle;
    context   IntIn  : Integer;
  where
    ObjOut = NullCycle;
  end Bar;

  phylum Baz
  with
    context   ObjIn  : Cycle;
  end Baz;

  ... AST operators in Part 2 go here ...

end Fibers;
```

Figure 10.1: A simple language description illustrating fibering (Part 1).

```
operator OpFoo : Foo is
  "foo" C1:Bar C2:Baz
where
  C1.CompIn = C1.ObjOut;
  C1.IntIn = 0;
  C2.ObjIn = C1.ObjOut;
end OpFoo;

operator OpBar1 : Bar is
  "bar1"
where
  object Obj : Cycle
  where
    Ref = OpBar1.CompIn;
    Int1 = 1;
    Int2 = OpBar1.IntIn;
  end Obj;
  OpBar1.ObjOut = Obj;
end OpBar1;

operator OpBar2 : Bar is
  "bar2" C1:Bar
where
  C1.CompIn = OpBar2.CompIn;
  C1.IntIn = OpBar2.CompIn.Ref.Int1 + OpBar2.IntIn;
  OpBar2.ObjOut = C1.ObjOut;
end OpBar2;

operator OpBaz : Baz is
  "baz"
where
  attribute Local : Cycle = OpBaz.ObjIn.Ref;
end OpBaz;
```

Figure 10.2: A simple language description illustrating fibering (Part 2).

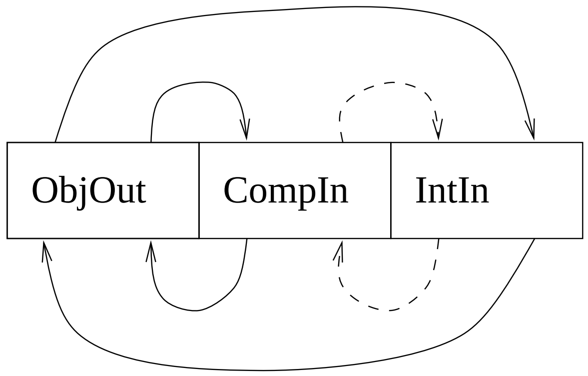


Figure 10.3: The summary dependency graph for phylum `Bar`, showing the induced dependencies among its attributes.

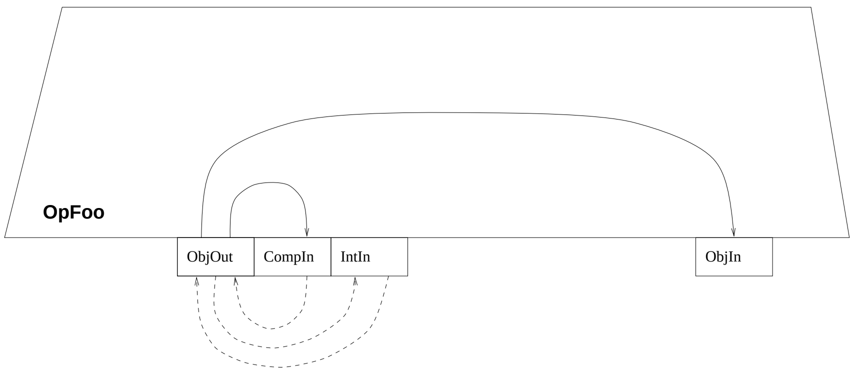
the resulting AG. The augmented dependency graphs computed during the initial DNC test are used to locate potential cycles that must be broken by the fibering transformation. The algorithm computes the strongly connected components (SCCs) of the augmented dependency graph for each AST operator, where each SCC may contain multiple overlapping cycles.

Since the dependencies among the attributes of `Bar` are themselves cyclic, clearly they will induce cycles when composed with those of any AST operator. In general, however, a cycle may occur in an augmented dependency graph even when the summary dependency graphs are all acyclic. Figure 10.4 shows the operators from the example of Figure 10.1 and Figure 10.2 in which potential cycles are discovered, and illustrates the dependencies graphically. The dependencies induced by composition with the summary dependency graphs are shown with dashed lines. Only those induced dependencies needed to complete all cycles are indicated, in order to reduce visual clutter in the diagrams. In each case, a single SCC is identified, whose nodes are as shown in the figure. In the sequel, we will continue to identify an SCC with the set of attribute occurrences it contains, i.e., its node set, with the understanding that the edges can be easily recovered from the augmented dependency graph if needed. The attribute occurrences are named using the compact notation employed by the compiler's diagnostic reports in which the child with which an occurrence is associated is identified by a numeric index, and the AST operator is left implicit. A child index of zero signifies that the occurrence belongs to the AST operator itself (i.e., the left-hand side of the production, in traditional grammatical terms), instead of one of its children.

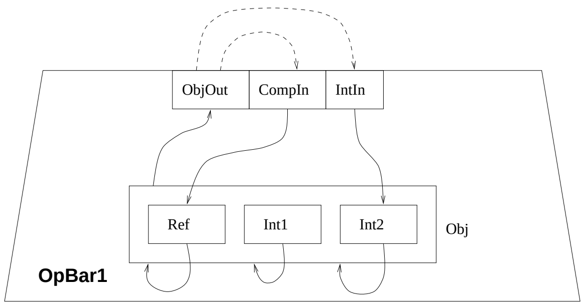
Figure 10.5 illustrates the ASTs of the example language schematically, in which cycles are readily apparent. Because of the approximation introduced by the use of the summary dependency graphs, potential cycles may be identified that cannot occur in any AST. For example, the attribute `0.IntIn` in `OpBar2` would seem to participate in a cycle involving the dependency of `0.IntIn` on `0.ObjOut` induced by the summary dependency graph for `Bar`. It is apparent from Figure 10.5, however, that this potential dependency cannot be realized in any AST.

Conventional attribute dependency analysis identifies cycles involving the attribute occurrences of each AST operator individually, examining the augmented dependency graph for each operator in isolation from the others. This is adequate if the goal is merely to determine if dependency cycles are present. To enable fibering, however, it is necessary to determine which attribute occurrences have instances that may appear on a common dependency cycle, even if the occurrences belong to distinct AST operators.

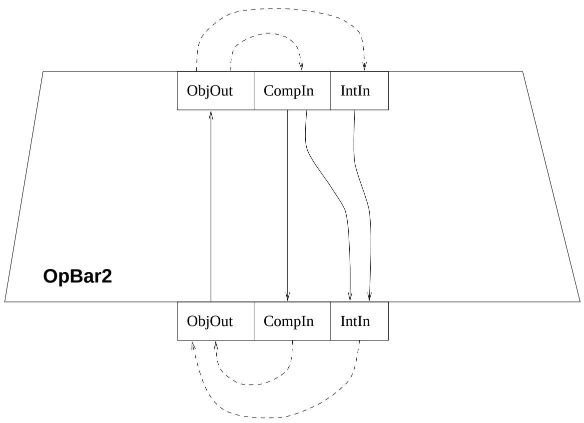
A node X is said to *adjoin* a node Y at child i in an AST if X is the i-th child of Y. In general,



$\{$ `1.ObjOut`, `1.CompIn`, `1.IntIn` $\}$



$\{$ `0.ObjOut`, `0.CompIn`, `0.IntIn`, `Obj`, `Obj.Ref`, `Obj.Int2` $\}$



$\{$ `0.ObjOut`, `0.CompIn`, `0.IntIn`, `1.ObjOut`, `1.CompIn`, `1.IntIn` $\}$

Figure 10.4: Potential dependency cycles (SCCs) in the example prior to fibering.

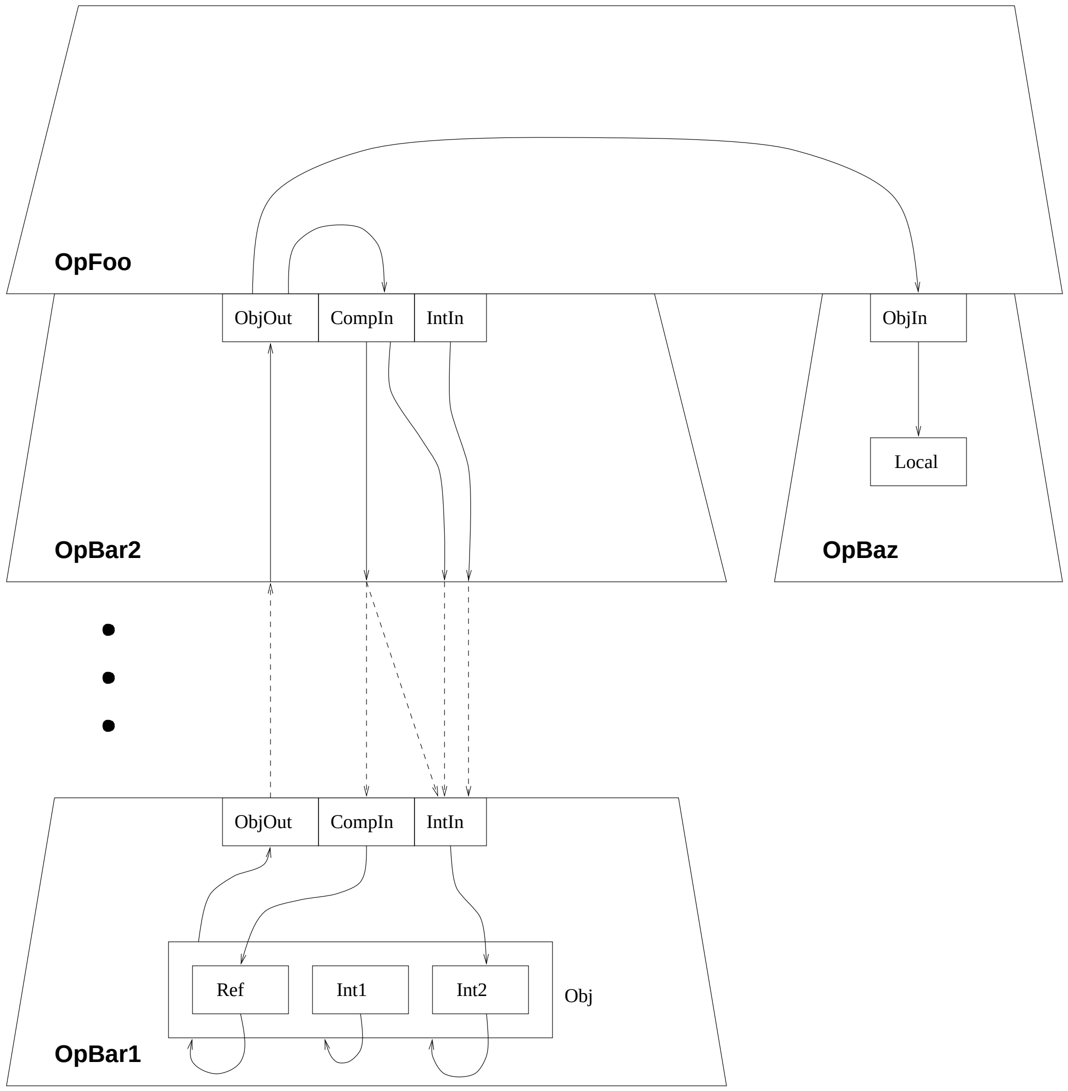


Figure 10.5: An AST with actual cyclic dependencies.

some instance of an AST operator A may adjoin some instance of an AST operator B at child i whenever the phylum of A and the phylum of the i-th child of B are identical. Let the relation xRy hold between two attribute instances x and y whenever x depends (transitively) on y and vice-versa, i.e., x and y belong to a common dependency cycle. Clearly, R is an equivalence relation. Let a and b be attribute instances of a node X, and let b and c be attribute instances of an adjoining node Y, where the instance b occurs at the interface of the two nodes and is thus shared between them. Then aRc if aRb and bRc. In general, if two dependency cycles within the attribute instances of distinct nodes of an AST share a common attribute instance, then all attribute instances in both cycles share a common cycle as well.

To permit a static analysis, we instead use a relation R' on attribute occurrences that conservatively approximates the dependencies obtaining among the underlying attribute instances. For occurrences x and y belonging to an AST operator A, $xR'y$ when x and y appear in a common cycle within the augmented dependency graph for A. Thus for each strongly connected component of the augmented dependency graph, $xR'y$ for every pair of occurrences x and y included therein. Furthermore, if $aR'b$ and $bR'c$, where a belongs to an AST operator A and c belongs to an AST operator B where an instance of A potentially adjoins B with an instance of b in common at the interface, we will assume that $aR'c$. Again, R' is an equivalence relation. The attribute occurrences whose instances may share a common dependency cycle are thus conservatively approximated by the (maximal) equivalence classes of attribute occurrences under R'.

Recall that the dependency analysis algorithm has previous computed for each AST operator the strongly connected components (SCCs) of its augmented dependency graph. We say that an SCC *potentially adjoins* another SCC if each contains an attribute occurrence that might give rise to a common instance in some AST. We define the relation *potentially connected* as the transitive closure of "potentially adjoins," and note that it is an equivalence relation. The equivalence classes of attribute occurrences that are potentially cyclically dependent may thus be conveniently represented as the equivalence classes of potentially connected SCCs. We denote the set of such equivalence classes for the analysis description under consideration by the symbol $\mathcal{C}$.

Each SCC is represented by a record that contains the set of SCCs that may adjoin it from above (as a parent) and below (as a child), as well as the AST operator and equivalence class to which it belongs. Each adjoining SCC is labeled with the child index at which it adjoins, and is thus represented as a pair. This additional structure is exploited in subsequent processing. In the example, the fibering algorithm finds three SCCs, one in each of the AST operators `Foo`, `Bar1`, and `Bar2`, as shown in Figure 10.6. All of the SCCs are potentially connected, thus they are all assigned to the same equivalence class. Hence $\mathcal{C} = \{$ `CLASS 1` $\}$, where `CLASS 1`$= \{$ `SCC 1`, `SCC 2`, `SCC 3` $\}$.

10.3.2 Dataflow Analysis

Once potential cycles have been identified, the fibering algorithm determines which attribute occurrences may access an object component via an object reference during the evaluation of its definition. Each such selection site is annotated with the set of component occurrences that might potentially be accessed. This determination is a relatively straightforward flow analysis problem, but the possibility that selections may occur within function calls, including calls to functions that are themselves object components, means that the flowgraph must be dynamically expanded during the analysis, as explained subsequently.

To simplify the flow analysis, we use a condensed flowgraph in which operations of the ADL language that are not relevant to objects and selections are elided. We thus do not consider the effect of any operation other than object instantiation or component selection beyond whether it may transmit an object received as a constituent of an argument as a constituent of its result value.

```
CLASS 1:

  SCC 1:

    Operator:  OpFoo
    Class:     1
    Above:     { }
    Below:     { < SCC 2, child C1 >, < SCC 3, child C1 > }

  SCC 2:

    Operator:  OpBar1
    Class:     1
    Above:     { < SCC 1, child C1 >, < SCC 3, child C1 > }
    Below:     { }

  SCC 3:

    Operator:  OpBar2
    Class:     1
    Above:     { < SCC 1, child C1 >, < SCC 3, child C1 > }
    Below:     { < SCC 2, child C1 >, < SCC 3, child C1 > }
```

Figure 10.6: The equivalence class $\mathcal{C}$ of potentially connected SCCs for the example, showing the potentially adjoining SCCs for each member.

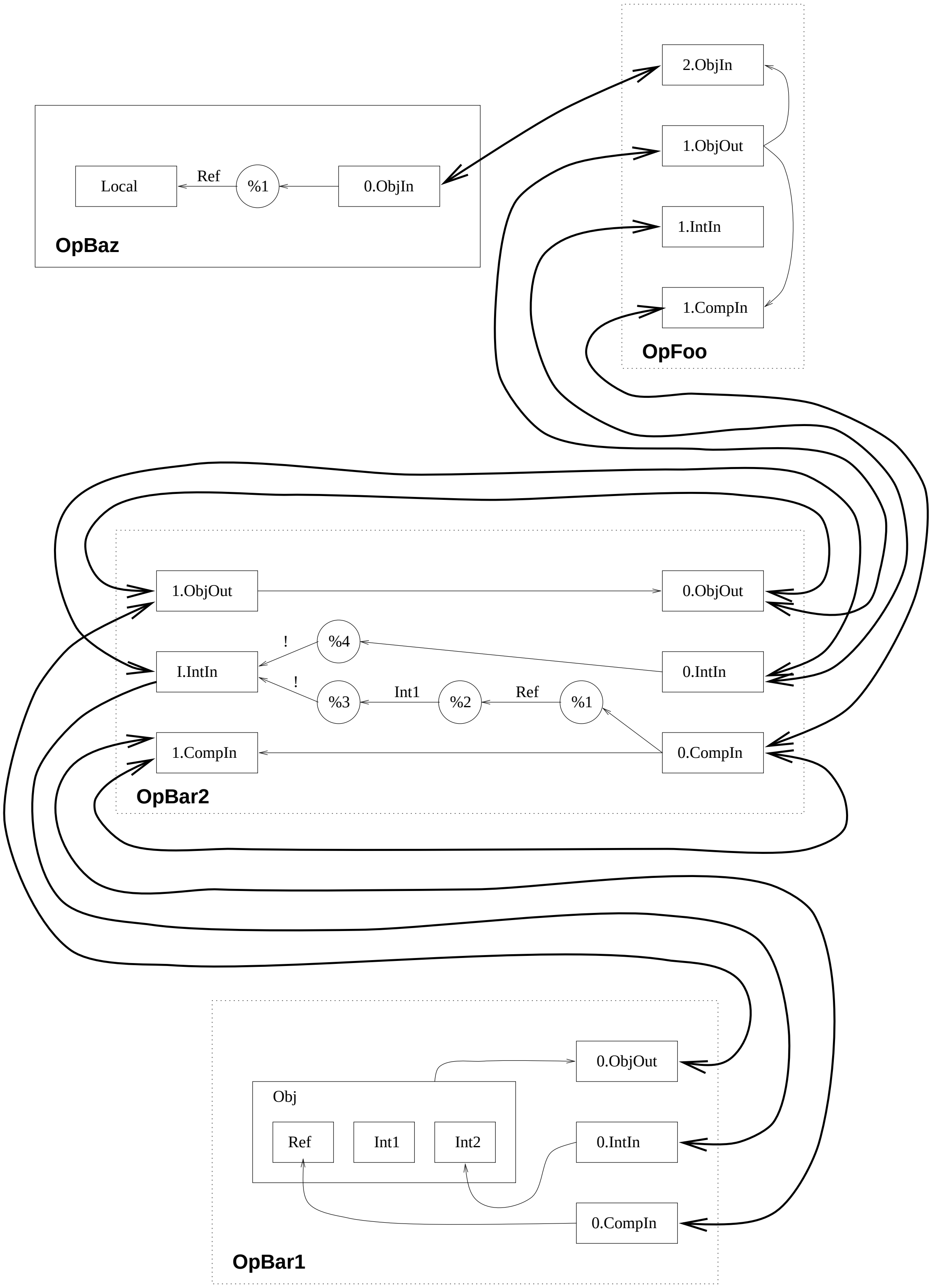


Figure 10.7: Flowgraph for the running example.

In particular, the components of aggregate data structures other than objects, including tuples and lists, are coalesced by the analysis.

Figure 10.7 shows the flowgraph for our example. The nodes of the flowgraph represent attribute occurrences (including components) and object instantiations, as well as intermediate expression results including variables, function calls, and component selections. A method invocation is treated as both a function call and the selection of a function-valued component. Figure 10.7, intermediate expression results are represented by circles and attribute occurrences by rectangles. There are three kinds of edges: (1) unlabeled edges, (2) labeled edges, which are annotated with the name of a component, and (3) blocking edges, which are labeled with a "!" character. Unlabeled edges may transmit any value. Labeled edges represent dataflow out from a selection, and transmit only the values which reach a like-labeled component of an incoming object. Blocking edges represent dataflow paths that cannot transmit an object reference, and in fact could be safely omitted from the flowgraph altogether. An example is the incoming arguments to an arithmetic operator. The blocking edges are included to make the correspondence between the flowgraph and the attribute definitions more apparent, as well as to reflect the fact that our implementation actually includes them, though for incidental implementation reasons.

Initially, a separate flowgraph is computed for each AST operator. Then, unlabeled copy edges are inserted to distribute values reaching each attribute occurrence (other than local attributes) to every corresponding occurrence of a potentially adjacent AST operator. This yields a complete flowgraph for the entire attribute grammar covering all possible runtime dataflow. These edges always occur in complementary pairs, and are thus represented in Figure 10.7 as heavy double-headed arrows. For clarity, we omit many edges that our algorithm would include but which are redundant due to transitivity.

Dataflow analysis proceeds iteratively using a worklist of flowgraph nodes. Each node is annotated with the objects that may reach it. When the outgoing edge is a labeled edge, the reaching objects are transmitted no further, but if any incoming objects have a component matching the label, objects reaching the component are propagated through the labeled edge.

Since function calls may perform selections, it is necessary to "look inside" the definition of the function when analyzing a function call. The function may itself call other functions. Rather than trying to compute a covering approximation of the dataflow within a function, the analysis simply expands the flowgraph at the point of a function call as if the function call had been expanded in-line at the source-code level. Clearly, this process leads to an exponential blow-up, and will fail to terminate in the presence of recursive functions, thus the expansion must be cut short using some further approximation. After experimenting with several more precise policies, we settled on expanding each function at most once within the portion of the flowgraph induced by each right-hand side of an attribute definition. All call sites invoking the same function are represented by the same node in the flowgraph, effectively merging their separate dataflow dependencies. This seems precise enough for our purposes, as borne out by our experience. The current flow analysis procedure is adequately fast, accounting for approximately twenty percent of the total analysis time for the full-scale language description for Modula-2 presented in Chapter 12.

Initially, the flow analyzer expands only those calls to functions that are called directly, i.e., not invoked via an object reference. Whenever a new object is found to reach a method invocation, the appropriate method definition is expanded. The dependencies induced by alternate methods reaching the method invocation are thus superimposed. If it were not for functional object components, flow analysis could be performed only for those attributes participating in a cycle. All we really need to know is which selections access components that are in the same cycle as the one whose definition contains the selection. Because we cannot tell which functions are actually called at a method invocation without knowing which objects reach it (even if it is not on any cycle), the flow analysis must be performed globally for all attribute occurrences in the language description. In practice, this has not been a problem. Figure 10.8 shows the objects that reach each node in the flowgraph of Figure 10.7.

Each flowgraph node contains a backpointer to the attribute occurrence whose definition induced the node, that is, the attribute occurrence whose evaluation would result in the execution of the computation for which the node represents the result. When every flowgraph node has been annotated with its reaching objects, it is a simple matter to examine each selection occurrence, including method invocations, and add the referenced components to the set of those selected within the evaluation of the appropriate attribute occurrence. In the example, all selections take place during the evaluation of attribute `Local` of AST operator `OpBaz` and attribute `1.IntIn` of AST operator `OpBar2`. The sets of potentially accessed components are {`NullCycle.Ref`, `OpBar1.Obj.Ref`} and {`NullCycle.Int1`, `OpBar1.Obj.Int1`, `NullCycle.Ref`, `OpBar1.Obj.Ref`}, respectively.

10.3.3 Fiber Reachability Analysis

At this point, the fibering algorithm has determined which object components are potentially accessed during the evaluation of the instances of each attribute occurrence. In order to assure that every instance of a fibered component is defined before it is accessed, the algorithm will introduce control attributes, their occurrences, and associated dependencies so as to link each such instance as a predecessor to the attribute instances that access the component remotely during their evaluation. In general, our strategy will be to collect the instances of a fibered component occurring within a cycle via synthesized attributes, and then, upon reaching a node that dominates all of the selection instances of the component, distribute them downward via inherited attributes to all occurrences of their selection sites. In preparation for the creation of the control attributes (to be discussed in the following section), the algorithm now annotates each AST operator that is potentially involved in a cycle with information that will permit the correct routing of the fiber dependencies. For each AST operator X, we compute the following values:

- `DEFINED_BELOW` The set of <*child*, *component*> pairs such that a subtree issuing from the indicated child of an instance of X may contain an object instantiation in which an instance of the component is defined.
- `SELECTED_BELOW` The set of <*child*, *component*> pairs such that a subtree issuing from the indicated child of an instance of X may contain an attribute instance whose evaluation may access an instance of the component.
- `DEFINED_ABOVE` The set of component occurrences such that a node ancestral to an instance of X may contain an object instantiation in which an instance of the component is defined.
- `REACHES_ABOVE` The set of component occurrences such that a node ancestral to an instance of X may have a descendent not in the subtree rooted at the instance of X in which the component is accessed remotely by a selection.

Recall that each SCC is labeled with the operator to which it belongs, an equivalence class of potentially connected SCCs, and the sets of SCCs potentially adjoining from above and below. Using the results of the flow analysis performed in the previous step, the fibering algorithm annotates each SCC with the set of component occurrences within the SCC accessed by any selection site within an SCC in the same class. Component occurrences are omitted that do not belong to an SCC of the same equivalence class as at least one of their selection sites. The fibering algorithm also annotates

```
OPERATOR OpFoo:

  1.IntIn            { }
  1.ObjOut           { NullCycle, OpBar1.Obj }
  1.CompIn           { NullCycle, OpBar1.Obj }
  2.ObjIn            { NullCycle, OpBar1.Obj }

OPERATOR OpBar1:

  Obj                { OpBar1.Obj }
  0.CompIn           { NullCycle, OpBar1.Obj }
  0.ObjOut           { NullCycle, OpBar1.Obj }
  0.IntIn            { }
  Obj.Ref            { NullCycle, OpBar1.Obj }
  Obj.Int1           { }
  Obj.Int2           { }

OPERATOR OpBar2:

  0.CompIn           { NullCycle, OpBar1.Obj }
  0.ObjOut           { NullCycle, OpBar1.Obj }
  0.IntIn            { }
  1.IntIn            { }
  %4                 { }
  %3                 { }
  %2                 { NullCycle, OpBar1.Obj }
  %1                 { NullCycle, OpBar1.Obj }
  1.ObjOut           { NullCycle, OpBar1.Obj }
  1.CompIn           { NullCycle, OpBar1.Obj }

OPERATOR OpBaz:

  0.ObjIn            { NullCycle, OpBar1.Obj }
  Local              { NullCycle, OpBar1.Obj }
  %1                 { NullCycle, OpBar1.Obj }
```

Figure 10.8: Objects reaching each node of the flowgraph (Figure 10.7) for the example.

each SCC with the set of object components that occur within it, as determined by a trivial scan of its constituent attribute occurrences.

The sets `DEFINED_BELOW`, `SELECTED_BELOW`, `DEFINED_ABOVE`, and `REACHES_ABOVE` are now computed for each AST operator. Since no component occurrence can occur within two distinct equivalence classes, and we are only concerned with selection sites that occur within the same equivalence class as their associated component occurrence, it suffices to compute the contribution of each equivalence class to these sets separately. Furthermore, since every SCC is labeled with the AST operator to which it belongs, the algorithm can use the SCCs as proxies for the underlying AST nodes, attaching the annotations to the SCCs rather than to the AST operators themselves. The annotations are computed by the algorithm shown in Figure 10.9 and Figure 10.10. The resulting annotated SCCs for the running example are shown in Figure 10.11.

10.3.4 Breaking Cycles and Inserting Control Attributes

For each object instantiation occurring in an SCC of an equivalence class in $\mathcal{C}$, the fibering algorithm now removes the dependency of the object on any of its component occurrences that appear on a cycle. (Because object components in ADL are full-fledged local attributes, it is possible for a component to appear in a cycle in which the containing object does not participate.) Recall that each component occurrence treated in this way is said to have been "fibered."

Having removed these dependencies, which previously assured that the component occurrences were defined before they were accessed, the fibering transformation must now add new dependencies that achieve the same purpose. These dependencies and their supporting control attribute occurrences, called *fibers*, link each fibered component occurrence as a predecessor to every selection site where it might be accessed.

The algorithm first considers the cases in which a selection site and a fibered component occurrence that it may access are contained in a common cycle, i.e., each belongs to an SCC of a common equivalence class. The reachability information computed in the previous step is used by the algorithm in Figure 10.12 and Figure 10.13 to generate fibers for these selection sites and component occurrences. The function `Create_Inh_Control_Attr` (`Create_Syn_Control_Attr`) creates an inherited (synthesized) control attribute occurrence belonging to the AST operator or child indicated by its first argument. The name of the attribute is of the form `UP_`*component* for a synthesized attribute and `DN_`*component* for an inherited one. If an attribute occurrence so named already exists, a new one is not created. Otherwise, a like-named attribute is added to the phylum to which the operator or child belongs, and an occurrence of the attribute is added to each AST operator and child position belonging to the phylum.

It is possible, indeed likely, that some attributes will lack predecessors or successors. This is a consequence of the fact that a node may actually need to transmit fiber dependencies only when it appears in certain tree contexts. In order to handle those contexts, the AST operator must provide the fiber dependencies, but in another context, the incoming or outgoing end of the dependency thread may be left unattached.

In order to minimize the proliferation of control attributes, when a selection site and a fibered occurrence that it may access cannot appear in a common cycle, a fiber is not generated. Instead, a single special fiber is created for each equivalence class that depends on *every* fibered component occurrence within the SCCs of the class. Every attribute occurrence outside of the class that depends on an attribute occurrence within the class is then made to depend as well on this special fiber, the *completion fiber* for the class. The routing of the completion fiber to those attribute occurrences, the "successors" of the class, can be performed by a simple variant of the fiber reachability analysis

```
% Compute DEFINED_BELOW, SELECTED_BELOW,
% DEFINED_ABOVE, and REACHES_ABOVE.

foreach class in 𝒞 do
  Compute_Fiber_Reachability(class)



global *worklist*



function Compute_Fiber_Reachability(class)
  *worklist* = nil
  foreach scc in class do
    Initialize_Fiber_Reachability(scc)
  until *worklist* == nil do
    scc = Worklist_Remove()
    Propagate_Fiber_Reachability(scc)



function Initialize_Fiber_Reachability(scc)
  foreach (child, above) in scc.ABOVE do
    foreach component in scc.COMPONENTS do
      unless (child, component) ∈ above.DEFINED_BELOW then
        above.DEFINED_BELOW =
                above.DEFINED_BELOW ∪ { (child, component) }
        Worklist_Add(above)
    foreach selection in scc.SELECTIONS do
      unless (child, selection) ∈ above.SELECTED_BELOW then
        above.SELECTED_BELOW =
                above.SELECTED_BELOW ∪ { (child, component) }
        Worklist_Add(above)
  foreach (_, below) in scc.BELOW do
    foreach component in scc.COMPONENTS do
      unless component ∈ below.DEFINED_ABOVE then
        below.DEFINED_ABOVE =
                below.DEFINED_ABOVE ∪ { component }
        Worklist_Add(below)
    foreach selection in scc.SELECTIONS do
      unless selection ∈ below.REACHES_ABOVE then
        below.REACHES_ABOVE =
                below.REACHES_ABOVE ∪ { selection }
        Worklist_Add(below)
```

Figure 10.9: Locating paths for control attributes: Fiber reachability analysis (Part 1).

```
function Propagate_Fiber_Reachability(scc)
  % Propagate DEFINED_BELOW and SELECTED_BELOW upward.
  foreach (child, above) in scc.ABOVE do
    foreach (_, component) in scc.DEFINED_BELOW do
      unless (child, component) ∈ above.DEFINED_BELOW then
        above.DEFINED_BELOW =
                above.DEFINED_BELOW ∪ { (child, component) }
        Worklist_Add(above)
    foreach (_, selection) in scc.SELECTED_BELOW do
      unless selection ∈ above.SELECTED_BELOW then
        above.SELECTED_BELOW =
                above.SELECTED_BELOW ∪ { (child, selection) }
        Worklist_Add(above)
  % In any child selects a component, then every sibling must propagate
  % definitions of that component upward to at least the current node.
  foreach (child1, selection) in scc.SELECTED_BELOW do
    foreach (child2, sibling) in scc.BELOW do
      unless child1 == child2 then
        unless selection ∈ sibling.REACHES_ABOVE then
          sibling.REACHES_ABOVE =
                sibling.REACHES_ABOVE ∪ { selection }
          Worklist_Add(sibling)
  % Propagate DEFINED_ABOVE and REACHES_ABOVE downward.
  foreach (_, below) in scc.BELOW do
    foreach component in scc.DEFINED_ABOVE do
      unless component ∈ below.DEFINED_ABOVE then
        below.DEFINED_ABOVE =
                below.DEFINED_ABOVE ∪ { component }
        Worklist_Add(below)
    foreach component in scc.REACHES_ABOVE do
      unless component ∈ below.REACHES_ABOVE then
        below.REACHES_ABOVE =
                below.REACHES_ABOVE ∪ { component }
        Worklist_Add(below)
```

Figure 10.10: Locating paths for control attributes: Fiber reachability analysis (Part 2).

```
SCC 1:
  Operator:       OpFoo
  Class:          1
  Above:          { }
  Below:          { < SCC 2, child C1 >, < SCC 3, child C1 > }
  Components:     { }
  Selections:     { }
  DefinedBelow:   { <child C1, OpBar1.Obj.Ref>,
                    <child C1, OpBar1.Obj.Int2> }
  DefinedAbove:   { }
  SelectedBelow:  { <child C1, OpBar1.Obj.Ref> }
  ReachesAbove:   { }

SCC 2:
  Operator:       OpBar1
  Class:          1
  Above:          { < SCC 1, child C1 >, < SCC 3, child C1 > }
  Below:          { }
  Components:     { OpBar1.Obj.Int2, OpBar1.Obj.Ref }
  Selections:     { }
  DefinedBelow:   { }
  DefinedAbove:   { }
  SelectedBelow:  { }
  ReachesAbove:   { OpBar1.Obj.Ref }

SCC 3:
  Operator:       OpBar2
  Class:          1
  Above:          { < SCC 1, child C1 >, < SCC 3, child C1 > }
  Below:          { < SCC 2, child C1 >, < SCC 3, child C1 > }
  Components:     { }
  Selections:     { OpBar1.Obj.Ref }
  DefinedBelow:   { <child C1, OpBar1.Obj.Ref>,
                    <child C1, OpBar1.Obj.Int2> }
  DefinedAbove:   { }
  SelectedBelow:  { <child C1, OpBar1.Obj.Ref> }
  ReachesAbove:   { OpBar1.Obj.Ref }
```

Figure 10.11: Results of fiber reachability analysis for the running example.

Create fibers for components and selection sites within a common class.

```
foreach class in C do
  foreach scc in class do
    Create_Fibers(scc)



function Create_Fibers(scc)
  operator = scc.OPERATOR
  foreach component in scc.COMPONENTS do
    if component ∈ scc.REACHES_ABOVE then
      % Propagate component dependency above if needed.
      succ = Create_Syn_Control_Attr(operator, scc, component)
      Create_Fiber_Dependency(component, succ)
      % Else we may need to propagate component below.  If already
      % available above, we will use the existing inherited attribute.
    else
      foreach (child, selection) ∈ scc.SELECTED_BELOW do
        when selection == component then
          succ = Create_Inh_Control_Attr(child, scc, component)
          Create_Fiber_Dependency(component, succ)
  foreach occurrence in scc.MEMBERS do
     'occurrence.EVALS_SELECTION' are the components that are potentially
      accessed during the evaluation of 'occurrence'.
    foreach selection in occurrence.EVALS_SELECTION do
      when selection ∈ scc.SELECTIONS then
        if selection ∈ scc.REACHES_ABOVE then
          % Use inherited attribute.
          pred = Create_Inh_Control_Attr(operator, scc, selection)
          Create_Fiber_Dependency(pred, occurrence)
          % No inherited attribute.  Look locally and below.
        else
          % Local definitions.
          foreach component in scc.COMPONENTS do
          when component == selection then
            Create_Fiber_Dependency(component, occurrence)
          % Definitions from below.
          foreach (child, component) ∈ scc.DEFINED_BELOW do
            when component == selection then
              pred = Create_Syn_Control_Attr(child, scc, selection)
              Create_Fiber_Dependency(pred, occurrence)
```

Function 'CreateFibers' continued in Figure 10.13 ...

Figure 10.12: Creating control attributes (Part 1).

... Function 'CreateFibers' continued from Figure 10.12

```
foreach (child1, component1) in scc.DEFINED_BELOW do
  if component1 ∈ scc.REACHES_ABOVE then
    % Propagate above if needed.
    pred = Create_Syn_Control_Attr(child1, scc, component1)
    succ = Create_Syn_Control_Attr(operator, scc, component1)
    Create_Fiber_Dependency(pred, succ)
  else
    % Not used above, so propagate downward from here.
    foreach (child2, component2) in scc.SELECTED_BELOW do
      when component1 == component2 then
        pred = Create_Syn_Control_Attr(child1, scc, component2)
        succ = Create_Inh_Control_Attr(child2, scc, component2)
        Create_Fiber_Dependency(pred, succ)
foreach (child, component) in scc.SELECTED_BELOW do
  % Propagate downward from parent if needed.
  when component ∈ scc.REACHES_ABOVE ∪ scc.DEFINED_ABOVE do
    succ = Create_Inh_Control_Attr(child, scc, component)
    pred = Create_Inh_Control_Attr(operator, scc, component)
    Create_Fiber_Dependency(pred, succ)
```

Figure 10.13: Creating control attributes (Part 2)

```
OPERATOR OpFoo : Foo

  1.CompIn             <- 1.ObjOut
  1.DN_OpBar1.Obj.Ref  <- 1.UP_OpBar1.Obj.Ref
  2.ObjIn              <- 1.ObjOut 1.UP_1

OPERATOR OpBar1 : Bar

  Obj                  <- Obj.Int1
  0.ObjOut             <- Obj
  Obj.Ref              <- 0.CompIn
  0.UP_OpBar1.Obj.Ref  <- Obj.Ref
  Obj.Int2             <- 0.IntIn
  0.UP_1               <- Obj.Ref Obj.Int2

OPERATOR OpBar2 : Bar

  0.ObjOut             <- 1.ObjOut
  1.CompIn             <- 0.CompIn
  0.UP_OpBar1.Obj.Ref  <- 1.UP_OpBar1.Obj.Ref
  1.DN_OpBar1.Obj.Ref  <- 0.DN_OpBar1.Obj.Ref
  1.IntIn              <- 0.IntIn 0.CompIn 0.DN_OpBar1.Obj.Ref
  0.UP_1               <- 1.UP_1

OPERATOR OpBaz : Baz

  Local                <- 0.ObjIn
```

Figure 10.14: Final attribute dependencies for the running example after fibering.

described previously in detail.[4] Figure 10.14 shows the attribute dependencies after the fibering transformation.

After adding the control attributes and their dependencies, the full OAG construction is attempted on the revised AG. If it succeeds, then an analyzer is produced, otherwise, the description is rejected. In the case of our example, the revised AG is in fact a member of the OAG class. Figure 10.15 exhibits an example AST and the dependencies among its attribute instances, showing how the fibering transformation has eliminated the cycles that appeared prior to fibering (i.e., Figure 10.5).

[4]For historical reasons, our implementation falls back in this case on a less clever routing procedure implemented before the full fibering mechanism, including the reachability analysis, had been worked out. It can in some circumstances create control attribute occurrences whose instances are always left "dangling" without any successors in every possible AST. It also introduces an unnecessary local control attribute occurrence for each equivalence class. These attributes, with names of the form `CMP_`n, may be seen in the compiler's diagnostic output. For clarity, these artifacts have been suppressed in the exposition and examples presented in this chapter.

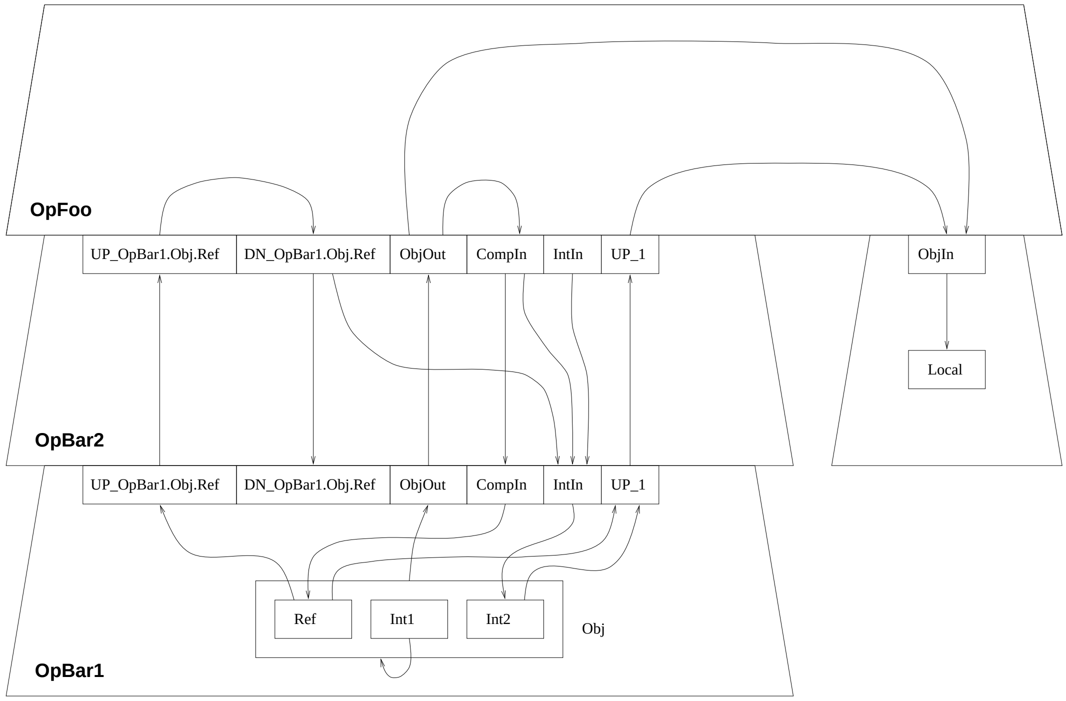


Figure 10.15: An example AST for the running example after fibering.

10.4 Refinements

As described above, the fibering algorithm fibers every component involved in any cycle. If an SCC contains more than one component, however, it will often be possible to break the cycles by fibering only a subset of the components. The fibering transformation imposes the restriction that *all* instances of a fibered component within a cycle must be evaluated before *any* of them can be accessed. This is in some ways a more restrictive condition than the original unfibered dependency of the object instantiation upon only a single component instance. For this reason, fibering a component unnecessarily may create new, spurious dependencies that were not present in the unfibered case, and which render the fibered description circular. To mitigate this problem, the algorithm only considers components for fibering that have been annotated with the pragma `delayed`. The keyword `delayed` is meant to suggest that the component may be evaluated at a later time than would normally be expected, i.e., after the remainder of the object instantiation in which it occurs. It is generally apparent to the description writer which component(s) should be fibered after examining and understanding the diagnostic reports from the attribute dependency analysis phase of the ADL compiler. The description writer proceeds iteratively, eliminating cycles by adding additional `delayed` pragmas.

Sometimes, a number of similar components play essentially the same role in the dependency analysis, and can thus be merged into an equivalence class that is scheduled as a unit. That is, all members of the class can be evaluated before all selections that access any of them. A class of such components may share a single fiber. An optional argument to the `delayed` pragma allows such equivalence classes to be specified. The pragma takes an identifier as an argument, which both identifies the class to which the attribute belongs and provides the shared fiber name that will be used in the diagnostic reports. Fiber sharing reduces the number of control attributes and their dependencies, resulting in shorter, more readable diagnostics and faster execution of the analyzer generator. Fiber sharing is used extensively in the language description for Modula-2 presented in Chapter 12.

Aside from the additional expressive power provided by fibering, our analysis algorithm collects additional information that may be helpful in diagnostics. The equivalence classes of potentially connected SCCs indicate more clearly the extent of a circular dependency than the SCCs in isolation. In the absence of spurious cycles introduced by the approximate nature of the summary dependency graphs, it would be straightforward to generate an actual example AST that exhibited a circular dependency. We have not investigated this topic further, however.

10.5 Related Work

Surprisingly, the problem that fibering is intended to address has been for the most part overlooked in the large literature on attribute grammars. This very likely is related to the fact that attribute grammars have been extensively studied in the academic literature, but are seldom employed in constructing real compilers, as noted by Waite [78]. Toy languages used in expository papers invariably assume declaration before use, and several well-known published attribute grammars for the "full-strength" semantic analysis of real programming languages implement languages with strong declaration before use requirements, e.g., Pascal in Kastens [46] and Ada in Uhl, *et al.* [73].

The problem and a proposed solution are presented by Farrow in a lamentably unpublished draft [22] describing an experimental extension to the Linguist [15] attribute grammar system. Farrow's approach is fundamentally different from ours, which we originally proposed in ignorance of his work. Farrow's treatment initially intrigued us, as it appears to be more general, permitting, for example, the fibering of tuples that are created dynamically within functions. Farrow claims, in fact, that his method of fibering attribute grammars is simply an application of a general method for statically converting a large class of functional programs requiring lazy evaluation to equivalent programs terminating under eager evaluation.

The essence of Farrow's method is to split the dependency upon an object into a separate dependency trace for each of its components (called a *fiber*) and for the reference to the object itself (the *base fiber*). Each selection depends on both the base fiber for the object and the fiber for the particular component selected. In naming his fibers, Farrow appeals to the metaphor of a dependency thread composed of still smaller fibers contained within. Fiber dependencies always follow alongside the original dependencies that they refine, unlike the dependencies that connect our control attributes, which are propagated along a path chosen in complete disregard for that taken by the original unfibered dependencies.

Since a component may represent an object reference, a component fiber for one object may itself function as a base fiber for some other object, and thus be further subdivided into fibers for its components. In general, each fiber has an associated *selection path* reflecting this recursive structure. The selection path is a sequence of component names denoting a sequence of selections by which the component occurrences upon which the fiber depends may be reached from the object reference associated with the (outermost) base fiber. The set of fibers derived from an original, unfibered attribute is in fact determined by the set of all such selection paths. This works as long as chains of references are of bounded length (and that this fact is apparent to the necessarily approximate static analysis), thus bounding the length of the possible selection paths. In the presence of potentially re-entrant data structures, however, we reach an infinite regress. It is thus necessary to approximate the true fiber dependencies by collapsing "similar" fibers into equivalence classes.

Farrow defines an abstract notion of fiber approximation, in which one fiber stands for a possibly infinite number of other fibers, and whose dependencies cover those of the fibers they approximate. Farrow only describes one concrete realization of such an approximation, the one used in Linguist. The idea is to let a *concrete* fiber in the approximation stand for every *true* fiber of which it is a prefix, identifying fibers with their selection paths. During the fibering process, in which fibers are subdivided further, subdivision stops when the name of a fiber would be extended with a component name that it already contains. This bounds the length of a fiber name, hence the number of concrete fibers.

Unfortunately, the draft is unclear on a number of crucial details, forcing us to attempt to reconstruct them. We were unable to come up with a satisfactory reconstruction. After several aborted attempts, brief discussions with Farrow, and extended discussions with John Boyland, we abandoned the approach and implemented a variant of our original proposal, as described here.

Indeed, while Farrow's underlying theory is appealing, the particular fiber approximation he proposes appears inadequate for our purposes. Our most plausible reconstruction of Farrow's method was unable to handle a description for a toy language similar to **Example**, but which employed a small refinement also used in our description for Modula-2. The language provides a type abbreviation mechanism, as in **Example**, that allows the programmer to introduce a new named type that is fully synonymous with its definition. It is thus possible, albeit an error, to create a pair of mutually supporting type definitions of the following form:

```
type foo = bar;
type bar = foo;
```

Upon encountering a named type when type-checking those definitions, the type-equivalence predicate attempted to follow the chain of renamings until a basic type or a type constructor was encountered. Clearly, the example above would lead to nontermination. Hence, in addition to the referent type, `Type`, the representation of a named type, `TypeEntity`, contained an additional component, `IsCyclic`. This component was initialized using the function `CyclicType` of Figure 4.6, and was later examined by the type-equivalence predicate.

It is easy to see why a prefix-based fiber approximation would fail on this example. It is essential that the dependency analysis note that a call to `CyclicType` does *not* access the `IsCyclic` component of any `TypeEntity` object. Since such an object may be reached after an arbitrary number of traversals of the `Type` component, however, no finite prefix approximation can distinguish the case in which, say, the final component selected is `IsCyclic` from the one in which it is `Type`.

Indeed, Boyland [9] claims that the prefix-based approximation as described by Farrow is flawed, and fails to account for all dependencies, according to the best understanding he could derive from the draft. He has also suggested a more powerful class of fiber approximations that could handle our example above, but has not, to our knowledge, worked out the details.

Chapter 11

The Implementation of Colander II

In the previous chapters, we have discussed the analysis definition language ADL and presented the key algorithms used in its implementation in a rather abstract way. In this chapter, we discuss the Colander II system as a concrete artifact.

Colander II consists of two components, namely, a stand-alone compiler for ADL and a runtime support package that is integrated with the host programming environment. Analysis descriptions are compiled into executable code to avoid the overhead associated with interpretive or table-driven approaches.

The compiled descriptions are dynamically linked and loaded into the host environment on demand as needed. The compiler supports multiple back ends, allowing a single program to process language descriptions descriptions for all supported environments. Currently, Colander II supports two host environments: Pan [3] and Ensemble [29]. Support for Pan is the most complete, and was implemented by the author. A port to Ensemble was performed by Nicholas Weaver. Unless otherwise noted, all references in this dissertation to the features and runtime performance of Colander II refer to the Pan version.

11.1 The Colander II Compiler

Rather than generate machine code directly, the Colander II compiler (**C2C**) generates code in the implementation language of the intended host environment, Common Lisp in the case of Pan, and C++ in the case of Ensemble. The resulting Lisp or C++ source code is then compiled to machine code by the appropriate compiler.

11.1.1 The Virtual Target Machine

To facilitate support for multiple targets, most parts of **C2C** deal exclusively with an abstract virtual machine called the *Virtual Target Machine* (VTM). Code for the VTM is highly reminiscent of a subset of Lisp, but contains a number of simplifying syntactic variations as well as many special-purpose operators related to AST traversal and change notification. VTM incorporates a simple object model that maps cleanly onto the facilities provided by most statically typed object-oriented languages, as well as the more dynamic ones such as the Common Lisp Object System (CLOS) [70].

In the VTM model, all functions are methods of an object. The program database object for a program document being edited is an instance of the analysis description class for the appropriate language. Global functions are methods of this object.

The translation of VTM to Common Lisp is in most cases a trivial transformation into equivalent Common Lisp forms or into calls to runtime support routines. The translation to C++ is somewhat more complicated but nonetheless straightforward, compiling to C++ code similar to that generated by other compilers for Lisp and Lisp-like languages that use C as an intermediate language, such as Kyoto Common Lisp [83].

The VTM has proven a very useful abstraction. The VTM to Common Lisp translation fits comfortably in a single source file, isolating the remainder of the compiler from any dependency on the target language. By providing an option to write out the VTM code in textual form, it was possible for another programmer working independently to produce the C++ port using a stand-alone VTM to C++ translator without requiring any changes whatsoever to **C2C** itself. An important benefit of VTM during debugging of the higher-level code generation algorithms is that the VTM code is much more concise and thus easier to read than the final Common Lisp or C++ code.

Although no such translators have been written, a direct VTM to machine-code translator would be no more difficult than for a very small subset of Lisp. Indeed, the absence of first-class closures would be a considerable simplification.[1] Since VTM code is only produced for type-correct ADL source programs, runtime tagging is needed only for storage-management purposes. In this way, VTM is closer to the intermediate representations used in compiling Standard ML than to Lisp.

11.1.2 Compiler Architecture and Implementation

The Colander II Compiler is written in Common Lisp, and runs as a stand-alone program. Parsing of the source code is performed by an LALR(1) parser, written in C and generated using **flex** [26] and **bison** [25]. The parser runs as a subprocess of **C2C**, and emits a sequence of abstract syntax trees, roughly one for each declaration or equation in the source file. These are internalized by the Common Lisp reader, which is connected to the subprocess via a Unix^{TM} pipe, and assembled into the complete AST.

The entire AST is maintained in memory during compilation, as the compiler proceeds in several passes over the internal form. A hash-link is maintained from the AST for each declaration to the symbol table object representing the declared entity. It was convenient to generate code for attribute equations, rules, and clauses while they were being type-checked. Unfortunately, the code for attribute references cannot be generated until scheduling has been completed (since some attributes will be temporary), which requires the attribute dependency information collected at the same time. We solved this and related phase ordering problems by introducing a number of special macro-like pseudo-VTM forms, such as symbolic attribute references, that are expanded during final VTM code generation. When the target-specific VTM code generator encounters one of these forms, it invokes a callback that performs the expansion. When VTM is output in textual form, a post-pass performs the expansion.

The compiler makes extensive use of the Common Lisp Object System, but is not written in a primarily object-oriented style. Inheritance is used mainly in the representation of the declared entities, in which entities of different kinds share much common structure. The AST remains in S-expression form and is traversed using a pattern-matching conditional construct similar to the Standard ML `case` construct, implemented as a Lisp macro.

[1] Funargs generated by `lambda`-expressions in VTM are only passed downward, and thus may always be stack-allocated.

We developed the compiler using the Allegro Common Lisp [27] implementation from Franz, Inc. The compiler is written almost entirely in portable Common Lisp, relying on implementation-specific extensions only for access to operating system facilities such as spawning processes and retrieving command-line arguments. It would not be difficult to port the compiler to another Common Lisp implementation with comparable facilities or an interface to foreign code written in a language such as C.

The compiler consists of approximately 14250 lines of Common Lisp code, not including the separate parser (in C), broken down as follows:

Parsing and Semantic Analysis	45%
Attribute Dependency Analysis	17%
VTM Code Generation	16%
VTM to Common Lisp Translation	9%
Internal Diagnostics	7%
Miscellaneous and Utilities	6%

Compiler performance is quite acceptable. The language description for Modula-2 described in the following chapter compiles to Common Lisp in approximately 6.5 minutes on a 64MB SPARCstation 10.[2] Originally, we produced a single Common Lisp file containing the translated code for an entire language description. For large language descriptions, the size of this file proved too much for the Common Lisp compiler to handle. We modified **C2C** to split the output among several smaller files that could be compiled separately. This approach actually results in a significant performance benefit because the files are small enough to process without invoking a costly global garbage collection.[3] It is much cheaper to start up a new Lisp image to compile the next file than to clean up after the old one. The **C2C** compiler generates a shellscript that is used to invoke the Lisp compiler on the series of output files, so the user need not be inconvenienced by the multiplicity of files. Compiling the Common Lisp code takes just over 8 minutes on a 32MB SPARCstation 2. For this reason, we usually load the Common Lisp code directly and run interpretively during debugging.

11.2 Runtime Support

In contrast to the single stand-alone compiler, the runtime support package must be implemented separately for each host environment. It consists of implementations for some of the more complicated ADL operators, such as term comparison, automatic storage management, and support routines for the maintenance of dynamic dependencies. In the Pan environment, storage management is inherited from the underlying Lisp runtime system. In Ensemble, a reference counting storage reclamation scheme is employed. Interestingly, ADL is amenable to reference counting even though circular data structures are permitted. From the standpoint of the storage manager, the components of an object need not be considered accessible via the references to the object, as every component is necessarily a local attribute of either an AST operator or the top-level of the description, and is thus already a root. Since every cycle within a data structure must include an object component, the heap is thus always acyclic from the standpoint of the storage manager.

[2] In contrast, compilation takes in excess of 15 minutes on a 32MB SPARCstation 2. In addition to the slower CPU speed, memory consumption is an issue. The compiler process grows to approximately 30MB while compiling the Modula-2 description with global garbage-collection suppressed.

[3] Our Common Lisp compiler employs a generational garbage collection scheme, in which most collections are small and non-disruptive. In contrast, a global collection may take 30 seconds or more, during which no useful computation is performed.

11.3 Retrospective Observations

Keeping the entire AST in core greatly simplified the construction of the compiler, facilitating our use of a multi-pass structure. An earlier version of the compiler used a single-pass structure, however, in which only the symbol table entries and fragments of VTM corresponding to the bodies of equations, rules, and clauses were retained. While adopting a multi-pass structure allowed us to relax certain constraints on the source language, the resulting compiler was much more complicated to write and debug. In retrospect, we question whether the added flexibility was worth the effort. Fortunately, we found that the extra storage consumption was not a problem on our hardware.

As observed earlier, compilation of the output of **C2C** to machine code using the target language compiler is a significant bottleneck during development of language descriptions. In the case of Pan, we can simply load the Lisp code directly and run it interpretively for testing purposes. This option is not available, however, when developing under Ensemble. The recent popularity of the Java^{TM} language, and the availability of a freely redistributable "just in time" compiler for the Java byte code instruction set [82], raises the intriguing possibility of generating Java byte codes directly from VTM source. The strongly typed, object-oriented, garbage-collected nature of the Java virtual machine is a good match for the semantics of VTM. Language descriptions compiled into such byte codes could be executed interpretively, compiled "on the fly" just before execution, or compiled off-line using tools developed or under development elsewhere, thus avoiding the cost of implementing equivalent functionality from scratch.

Chapter 12

Gaining Experience: Analyzing Modula-2

In Chapter 4, we presented an analysis description for an artificial toy language in detail. Real-world programming languages, designed to solve real problems rather than to clarify an academic argument, often pose severe challenges for compiler-generation tools based on idealized "textbook" principles. In this chapter, we report the results of an experiment undertaken to discover the strengths and limitations of our approach in a realistic setting.

12.1 Static Semantics for Modula-2

We have implemented a static semantic analyzer for the programming language Modula-2. We chose to implement Modula-2 for several reasons:

- Modula-2 has a rich and complex scoping discipline.
- A grammar was available for Modula-2 which accommodated limitations in the incremental parser we rely on, which cannot handle situations requiring feedback from semantic analysis to the parser, e.g., the use of type names defined via `typedef` in C.
- Modula-2 was used as a full-scale example for the old Colander system described by Ballance [4], allowing us to build a similar description for Colander II.

Our coverage of the language is similar to the description developed for Colander by Ballance, omitting the contents of the `SYSTEM` module, as well as the new types `LONGINT` and `LONGREAL` introduced in later editions of the Modula-2 report. We were also forced to introduce some unfortunate restrictions on the use of constant expressions due to limitations in ADL and its statically scheduled evaluation model, which will be discussed subsequently.

We generalize the language in one important way, placing no restrictions on the order in which declarations appear within each scope. According to the report, a name may not be used within another declaration before the name has been declared unless the use is embedded within a statement. Thus, for example, a procedure call, appearing as a statement within a procedure definition, may reference a procedure not yet defined, but a type, appearing within a type declaration, may not. The principal effect of this restriction is to prohibit the definition of ill-founded circular types and constants, for which we provide a separate check. We lifted the declaration before use requirement because such restrictions are considered by many authors to be an artifact of conventional sequential batch-mode compilation, and an inappropriate restriction in an interactive language-based editing environment. Hedin [32], for example, claims that even when the language definition requires definition before use, the restriction should be relaxed by the editor, and conformance restored when exporting source text by topologically sorting the declarations. Furthermore, a direct encoding of the usual method of testing for declaration before use, i.e., introducing declarations into the environment one at a time as they are processed, leads to threading the binding environment attribute through the declarations. To permit efficient incremental evaluation, the recommended strategy is to manage the binding environment as if there were no restrictions, and then to perform a separate test for declarations that are out of order. This approach also leads to clearer diagnostics. Unfortunately, we discovered that such common language features as named symbolic constants and unrestricted declaration order are problematic in any attribute grammar that is to be evaluated by a statically scheduled visit-sequence evaluator, quite independently of incrementality concerns.

The Modula-2 report is unclear on many points, and is not an exemplary defining document. We relied on Blaschek and Pomberger [7] and Cornelius [14] for clarification on a number of points. We will note where our implementation deviates from our best assessment of the intent of the report or common practice as reflected in those other authorities.

12.2 Implementing Analysis of Modula-2

Our analyzer for Modula-2 exports a single relation, `Error`, associating AST nodes with the error messages that apply to them. In its essential style and structure, the Modula-2 description resembles the one for **Example** developed earlier. The text of the description exceeds 5000 lines in length, precluding a detailed walk-through such as that we presented for **Example**. We will thus only highlight important design decisions, along with specific difficulties and their resolution. The complete text of the Modula-2 description is included as Appendix C.

12.2.1 Compilation Units

In the general case, a Modula-2 program is composed of multiple compilation units. Conventionally, the source code for each such unit is stored in a separate file. Since Colander II currently creates a distinct analysis database for each source file opened for editing, with no provision for communication, the analyzer would be unable to verify inter-unit consistency if each unit were stored in a separate file. Our analysis is thus defined to process a single source file, in which all of the compilation units of the program are included in an arbitrary order.

In Colander, each compilation unit is contained in a separate file, but the required communication is provided simply by allowing all source files using a common language description to share the global context of the description. There is no notion of a "system" of related files composing a single program. A more satisfactory solution is provided by the Segmentable Attribute Grammars of Micallef and Kaiser [53], which allow separately stored and editable subtrees to be dynamically "grafted" onto another tree for attribution.

12.2.2 Scoping and Name Resolution

Modula-2 defines a complex scoping discipline in which many language constructs establish binding environments, each with idiosyncratic name-resolution rules. *Definition modules* declare a set of named entities, which are then provided with implementations in an *implementation module*. Any

```
class ImportedName isa Entity
with
    attribute RefersTo : BindingStatus
     delayed AllImportsResolved;
requiring
  attribute Imported : BindingStatus
     delayed AllImportsDeclared;
where
  RefersTo = StripImportProxies(Imported);
end ImportedName;
```

Figure 12.1: Proxy for imported names.

module may import the declarations provided by a definition module by reference to the module name. An implementation module implicitly imports its corresponding definition module. *Program modules* are similar to implementation modules, but do not have a corresponding definition module, and contain the main program. In addition to these module types, which are all separate compilation units at the top-level of the program, *local modules* may be nested within any scope. Local modules control visibility using somewhat different rules than other kinds of modules. Procedures and functions also introduce new scopes, following the familiar Algol nesting model. The components of a record variable may be "opened" for access within the scope of a `with` statement, as if they had been declared as individual variables.

Explicit lists of imports pose a problem, due to the fact that local modules may be mutually recursive. In the context in which they appear, the imports function as declarations. The imported entity, however, defined in another module, may depend on the entity being defined. This potential circularity renders the definition circular, prior to fibering. For fibering to work, however, every cycle must be mediated by at least one object. Because a trivial ill-founded cyclic definition is possible (and must be checked for later), we cannot rely on the object representing the imported entity to fulfill this role. We thus introduce a special entity type to represent the imported name, which serves as a proxy for the imported entity, and contains an indirection to it. In effect this is no different than the objects we we must introduce to represent named types, but, in that case, the explicit representation of a named type as distinct from its referent seems semantically well-motivated, and not merely a device to make fibering work. The class definition for the proxy entities is shown in Figure 12.1. To avoid the overhead of repeatedly dereferencing chains of such proxies, each proxy contains a pointer to its ultimate referent which is initialized as early as possible, subject to the constraints of our static scheduling algorithm. Explicit exports appearing in local modules are handled similarly, as if the surrounding context had performed an import.

The binding environments for all constructs introducing scopes are represented as subclasses of the class `Context`, declared as shown in Figure 12.2, and follow a common protocol for name resolution. The `Context` class is very similar to the `Contour` class that we developed for **Example**. The currently visible binding for a name is returned by the `VisibleBinding` method. Two versions of the local bindings are maintained. The first, `LocalBinding1`, is used during the processing of declarations, and may contain import proxies. When the ultimate referents of the proxies have been determined, `LocalBinding2` is constructed as a copy of `LocalBinding1` in which all import proxies have been replaced with their referents. In this way, it is not necessary for subsequent name resolutions to chase proxy indirection chains, thus reducing the number of dynamic selections that

```
class Context
with
  function   VisibleBinding(String) -> BindingStatus
                maintained,
                delayed VisibleBinding;
  function   LocalBinding1(String) -> BindingStatus
                maintained,
                delayed LocalBinding1;
  function   LocalBinding2(String) -> BindingStatus
                maintained,
                delayed LocalBinding2;
  function   Declarable(String, Entity) -> Boolean;
  function   Redeclares(String) -> BindingStatus;
  attribute  IsInLoop : Boolean;
  attribute  InProcedure : Context
                delayed InProcedure;
requiring
  % nothing
where
  ...
end Context;

class TopContext isa Context
with
 function LiteralBinding(String) -> BindingStatus
                delayed LiteralBindings;
requiring
  relation   Binds(String, Entity)
                maintained;
where
  ...
end TopContext;
```

Figure 12.2: The class `Context`.

must be registered at runtime. The method Declarable determines if a declaration constitutes an illegal duplicate declaration. The method Redeclares is applicable only within an implementation module, and determines which declaration inherited from the corresponding definition module is to be superseded, if any. Context objects are also used to record nesting within procedures and functions, and within loops, to assist in the analysis of the return and exit statements.

Enumeration literals, defined in the declaration of an enumeration type, are treated like any other named constant, and bound in the context in which the type definition appears. When the name of an enumeration type is imported, however, Modula-2 requires that the literals be implicitly imported as well. In ADL, we cannot treat such implicit imports as declarations, as we cannot instantiate any objects (e.g., import proxies) to represent the literals. This follows as a consequence of the static instantiation strategy for objects. We thus maintain another set of local bindings, represented by the method LiteralBinding, which is derived by examining the referent of each name imported into the context and collecting enumeration literals. During name resolution, VisibleBinding checks both LocalBinding2 and LiteralBinding. A consequence of this strategy is that declarations appearing locally may silently shadow imported enumeration literals. While this may be arguably useful behavior, it appears that the conventional interpretation of the implicit import rule is that it should have the force of a declaration, producing an error message in such cases.

12.2.3 Types and Typechecking

In most cases, an expression represents a value. In some contexts, however, phrases that are grammatically expressions play a different role. Specifically, the arguments to a procedure are syntactically expressions, but may denote a storage location in the case of a reference (var) parameter, or even a type name in the case of the standard procedure VAL. Each expression is thus given an attribute Mode which distinguishes these roles. The type of the Mode attribute is the following:

```
datatype ExprMode is EmVAR       % storage location
                   | EmVAL       % value known at runtime
                   | EmTYPE      % type name
                   | EmCONST     % constant value known at compile-time
                   | EmUNKNOWN   % unknown due to error
```

In the event that the expression denotes a compile-time constant, another attribute, Value, provides additional information. Its type is:

```
datatype ConstValue
     is CvUNKNOWN           % no further information
      | CvILLFOUNDED        % defined by cyclic named constant
      | CvNILPTR            % the standard constant NIL
      | CvPROCVALUE         % a procedure value
      | CvSTRING(String)    % a string literal with given value
      | CvINTEGRAL(Integer) % integer or ordinal with value
      ;
```

Our description maintains only the information that is relevant to performing required semantic checks, unlike a full compiler which would need more complete information for code generation. It does not maintain any information concerning the value of REAL constants. Since no required compile-time checks involve real values directly, this choice appears to be acceptable. If the standard function TRUNC is permitted as a static compile-time function, however, it is possible that, say, the bounds of an array might be derived from the value of a real-valued constant expression. (It is not

```
class TypeObj isa OperandMatchResult
requiring
  attribute Shape : TypeShape;
end TypeObj;

datatype TypeShape     is TsUNKNOWN           % unknown due to error
                        | TsINTEGER
                        | TsCARDINAL
                        | TsCHAR
                        | TsREAL
                          % object containing bounds information
                        | TsSUBRANGE(SubrangeObj)
                          % list of enumeration literals
                        | TsENUMERATION([Constant])
                          % index type, element type
                        | TsARRAY(TypeObj, TypeObj)
                          % element type
                        | TsOPENARRAY(TypeObj)
                          % contour containing field/name bindings
                        | TsRECORD(RecordContext)
                          % base type
                        | TsSET(TypeObj)
                          % argument list
                        | TsPROCEDURE(ArgSpecList)
                          % argument list, result type
                        | TsFUNCTION(ArgSpecList, TypeObj)
                          % referent type
                        | TsPOINTER(TypeObj)
                        | TsOPAQUE
                          % symbol table entity for referenced type
                        | TsABBREV(NamedType)       % named type
                        ;
```

Figure 12.3: The representation of types.

clear from the language definition precisely which operations are intended to be evaluable at compile time, however.)

Every expression has an associated type, represented by the `Type` attribute. The type-equivalence rule in Modula-2 is fundamentally equivalence-by-name, though with many exceptions and special cases. In order to distinguish distinct types possessing isomorphic structure, we represent types using objects. These objects, of class `TypeObj`, contain a single component, `Shape`, a term type encoding the structure of the type. These definitions appear in Figure 12.3.

For most types, the representation of the type structure is straightforward. Subrange types pose a problem, however, as the bounds of the subrange are constant expressions, whose type may potentially (though erroneously) depend on the subrange type being defined. Resolving this circularity requires that the bounds of the subrange be represented as object components so that fibering may apply. We thus use an auxiliary object, of class `SubrangeObj`, as follows:

```
class SubrangeObj
requiring
  attribute Type : TypeObj;
  attribute Min : ConstValue
        delayed SubrangeBounds;
  attribute Max : ConstValue
        delayed SubrangeBounds;
end SubrangeObj;
```

There is a further complication with subrange bounds which reveals one of the more serious limitations of static analysis in our system. In general, the type of an expression may depend on the type of a variable, and the type of a variable may be a subrange. Thus a cyclic attribute dependency is inevitable if a general expression is permitted as a subrange bound, even if that expression is restricted to have a constant value. Fibering does not help in this case, as it is necessary to actually evaluate the expression, not just transmit an object reference unexamined. We are thus forced to create two essentially identical versions of the expression subgrammar, one for use in contexts where a constant expression is required (`CExpr`), and another for the general case (`Expr`). The constant expression subgrammar need not access the `Type` component of variable entities, as the mere presence of a variable name within the expression is sufficient to indicate an error. The type of a constant expression will not depend on the types of variables, thus once the `Type` component has been fibered, the circularity will be eliminated. As it turns out, we replicate only a small part of the expression subgrammar for constant expressions, resulting in a severe language restriction, due to a more fundamental problem with the processing of named constants under a static scheduling discipline and without declaration order restrictions. This issue will be discussed shortly.

Similarly, we cannot simply use the definition of a named type in place of the name, as the named type may itself potentially depend on the current type, requiring the use of an indirection to avoid an unfiberable trivial cycle. In this case, however, the entity representing the named type serves this role. It is, in any case, advisable to retain information on the use of named types, as this information may be helpful in error diagnostics. Named types are represented by instances of the `NamedType` class:

```
class NamedType isa Entity
requiring
  attribute Type     : TypeObj
        delayed AllNamedTypesResolved;
  attribute IsCyclic : Boolean
        delayed AllTypeCycleChecks;
  relation  HasLiteral(String, Constant)
        delayed NamedTypeEnumLiterals;
end NamedType;
```

The component `IsCyclic` is initialized after the definitions of all named types (within a given contour) have been completed. During type-checking, several functions must traverse potentially cyclic type descriptor structures without falling into infinite regress should an ill-founded type be encountered. Since the full check for circularity is expensive, we perform it once for each named type and cache the result. Cycle-checking is performed using a trail in a manner analogous to our analyzer for **Example**, except for the handling of record fields, which requires some way to step through the record field definitions. The `RecordContext` class, which holds the bindings of field names to their associated field entities, is given an additional functional attribute `CyclicFields` that takes a trail argument and carries out the cycle-check for the fields of the record. The cycle-check then delegates responsibility to this method when a record type is encountered. For the name of an enumeration type, the relation `HasLiteral` enumerates the names and constant entities for its literals. These are used in the definition of the `LiteralBinding` attribute described in the previous section, which handles the implicit import of the literals when the name of an enumeration type is imported.

Assignable storage locations such as variables, parameters, and record fields, are represented as instances of subclasses of the class `Location`. Their shared, variable-like behavior is captured in the common superclass, while allowing, for example, parameters to have a transmission mode annotation. Objects of class `Location` are instantiated directly to represent the result of operations such as subscripting, which denote unnamed assignable locations.

12.2.4 Declarations and Named Constants

Beyond the issues previously discussed, the processing of most declarations is entirely analogous to their treatment in **Example**. The declaration of named constants, however, poses a problem which is intractable within our current statically scheduled evaluation framework. Without restrictions on declaration order, the definition of any named constant may depend on a reference to any other constant declared in the same scope. We could, in principle, create an isomorphic copy (as an ADL data structure) of the expression defining each constant, then traverse the copy within an out-of-line function to perform the evaluation. References to other named constants could be resolved into direct links to the referent definition in the same manner as for type descriptors. This would yield a potentially cyclic expression graph, requiring the use of a trail to detect and report cycles as errors. This approach is completely antithetical to the spirit of attribute grammars, however, in which semantic functions should be small and limited in the amount of processing they perform. *Any* computation over a tree can be computed in a single pass by building a copy of the tree and doing all the processing in a semantic function at the root, but this approach clearly defeats whatever leverage the AG formalism might provide, as well as rendering incremental evaluation methods totally ineffective.

The difficulty is that the required evaluation order for a set of constant definitions depends in a crucial way on the specific usage of constant names within the definitions of other constants. While cyclic ill-founded constant definitions are illegal, the possibility of their construction cannot be ruled out from the structure of the statically analyzable attribute dependencies.

We need a mechanism combining two ideas: First, the attributes of constructs such as constant definitions must be evaluated in a demand-driven way, adapting the evaluation order to the specific pattern of references between the definitions. Second, the mechanism must detect the actual presence of a cycle dynamically, and handle a cycle as an exceptional event, e.g., by computing a reserved error value as the value of an affected attribute. By relying on a builtin mechanism to detect and recover from cyclic dependencies, it would then be possible to ignore cycles at static analysis time, confident that they could not result in inconsistency at runtime. We propose a special conditional construct which would request the value of an attribute while providing an error value to return if a dynamically detected cyclic dependency precluded the evaluation. An instance of this construct, or a `delayed` object component, would then be required within each potential cycle identified by static analysis. This mechanism would thus provide a second method of resolving circularities in addition to fibering.

We made a partial attempt at prototyping a solution along these lines by using existing ADL mechanisms to simulate demand-driven evaluation within the constant expression subgrammar. The `Type` and `Value` attributes for AST operators within this subgrammar are functional attributes that compute the appropriate type and value when called. Each takes a trail parameter, analogous to that used in the detection of cyclic types, which allows the functions to return a reserved "ill-founded" value when needed. The attributes of the named constant entities are defined similarly:

```
class Constant isa Entity
requiring
  function Type(TrailType) -> TypeObj
    delayed ConstantTypes;
  function Value(TrailType) -> ConstValue
    delayed ConstantValues;
end Constant;
```

Every cycle in a constant definition must include a named constant. Completing each cycle via an object method invoked remotely circumvents the circularity test, which does not attempt to restrict mutually recursive method calls.

We implemented this approach for a subset of the expression sublanguage. It is very awkward to access the values of the attributes within a constant expression, e.g., for checking the types of operands to an arithmetic operator, as the functional attributes must be called afresh at each point that we wish to examine the value. In order to prevent an exponential blowup in the number of function calls, the functions must be cached, and the existing caching mechanism will not perform well with large trail values as arguments. We simply restricted constant expressions to the simple case of a literal or a reference to another constant, and did not implement any operations on constant expressions other than sign inversion (unary minus).

12.3 Lessons from the Implementation of Modula-2

In general, we found the facilities of ADL a pleasure to use. Our experience might be summarized by saying that most of writing a language description was very easy, but that the few difficult parts were nearly impossible.[1] In this section, we review the problems that we encountered.

12.3.1 Dynamic Scheduling

The difficulty with named constants came as a complete surprise to us, as we found no mention of such a problem within an extensive literature on attribute grammars. Yet it appears endemic to any attempt to implement two rather common language features, named constants and unrestricted declaration order, within a statically scheduled framework. Indeed, regardless of the scheduling method, the resulting AG would be circular according to the classical and generally accepted definition. This problem was the major one, and the only showstopper.

We have suggested a solution, and hand-crafted a "mock-up" to make an initial assessment of its feasibility. The result, however, brings heavyweight mechanisms (function caching, trailing) to bear, with unacceptable runtime cost. What is really needed is a way to schedule the actual node visits dynamically, so that processing that normally takes place within a visit function, such as storage of attribute values in the tree (if needed) and performing assertions into collections, could be done at the same time. In the case of constant expressions, for example, a single visit to each named constant and each expression and subexpression of its definition suffices to compute all relevant attributes, or to detect any cycles present. Our incremental evaluation methods, however, depend on static scheduling, and would have to be replaced in these cases with other methods of an unspecified nature.

[1] This is not an uncommon situation with tools that rigidly adhere to powerful high-leverage principles at the expense of pragmatic flexibility. We will have more to say about this in the following chapter.

12.3.2 Fibering

Fibering did not work as well as we had hoped, forcing us to treat the elimination of dependency cycles as an iterative, intuitively guided, trial-and-error process similar to removing LALR(1) conflicts from a grammar to be processed by a parser generation tool such as **bison**. The multitude of control attributes introduced by fibering complicated the process of removing the so-called "Type 3" circularities [47] detected in the final step of the OAG evaluator construction. These "circularities" are artifacts of the greedy algorithm used by the OAG evaluator construction to determine a total ordering on the attributes given the partial order implied by the dependencies in the AG. As in all evaluator generators based on OAGs, the user must coax the algorithm into making a feasible choice by further constraining it with additional and otherwise useless attribute dependencies. This process requires building some intuition about the attribute dependencies in the program, generally with the help of the dependency analysis diagnostics. The large number of control attributes and their dependencies introduced during fibering make the diagnostic dumps very difficult to interpret, however, especially when the attribute causing the problem is itself a control attribute.

Another failing of the fibering algorithm is that it was necessary to fiber many more components than would require backpatching in a hand coded compiler. The problem is that each equivalence class of adjoining cycles (representing an SCC in the dependency graph) is viewed as a unit, when in fact it may contain a number of logically disjoint cyclic paths that cannot be distinguished due to the approximate nature of the initial (unfibered) dependency analysis. Every cycle arising in Modula-2 involves a reference to a name, assuring that every cycle contains the binding environment. Thus fibering identifies only a single SCC in the entire description, resulting in the assumption that every component involved in an SCC depends on every other such component within the binding contour in which it appears.

We conjecture that it would be possible to get much better dependency information if fibering were done one component at a time. Fibering a component to remove one cycle may reveal that other cycles discovered during the initial analysis are spurious. Thus the analysis should be repeated, and another component fibered, and so on until no more are left. The resulting fibering order, however, would in fact *be* an ordering on the relative dependencies between the components, and it is not clear how it would be determined, other than by user intervention.

12.3.3 Objects and Non-local Dependencies

We found it necessary to use objects in some cases in order to exploit their possession of a unique identity, but where the maintenance of non-local dependency links was not appropriate. That is, we would prefer that they be treated like ordinary tuples for analysis and evaluation purposes. It would be helpful if the maintenance of nonlocal dependencies could be selected by a pragma, perhaps for each component individually.

12.3.4 Data Types and Operators

In order to handle compile-time real arithmetic, we need support for fractional numbers. Since floating-point "reals" are in fact just rational numbers, we suggest that rational numbers, e.g., as supported by Common Lisp, be included. This would avoid making assumptions about target-environment floating point formats in ADL.

Chapter 13

Evaluation and Future Directions

13.1 Summary of our Research Contributions

This dissertation makes three major contributions. Though they contribute synergistically to the complete Colander II system, each stands alone as an independent contribution applicable in a more general context as well.

We have developed a new formalism for expressing executable specifications of static semantics. Our metalanguage melds the advantages of traditional attribute grammars, including amenability to extensive generation-time analysis, with the expressiveness and client-independence characteristic of Ballance's Logical Constraint Grammars. Our formalism allows much more of the incrementality inherent in a particular analysis problem to be exposed within the formalism itself, where it can be exploited automatically by our implementation. In contrast, traditional attribute grammars conceal much of the interesting computation within "black box" semantic functions.

We have developed incremental analysis algorithms tailored to our formalism that exploit its distinctive features. These features include object-valued and function-valued attributes, which allow us to automatically generate incremental analyzers that handle long-distance dependencies and aggregate attributes efficiently. Our methods allow unusual freedom to control the granularity of incremental evaluation, allowing performance tradeoffs to be chosen as demanded by the needs of the application rather than the *a priori* requirements of the algorithms. Our methods are also distinguished by the simplicity of the required runtime machinery and their minimal reliance on out-of-line library support, making it practical to compile the evaluators to low-level executable code. While developed to address the needs of our specification formalism, our incremental evaluation algorithms do not make essential use of attribute grammar concepts, and are applicable to other evaluation methods based on traversal and annotation of an abstract syntax tree.

We have developed a static analysis and transformation on attribute grammars that accommodates a useful class of circular attribute dependencies. The method automates the "backpatching" method often used in hand-coded compilers. The transformation is applicable to attribute grammars in general, and has no special dependence on incremental evaluation or the particulars of our specification metalanguage.

13.2 Open Issues in Colander II

Unfortunately, the Colander II system as a whole has only partially met its design objectives. In this section, we summarize those areas in which further exploratory research or engineering effort is needed.

While suitable for expressing a wide range of programming language concepts, realistic programming languages demand additional facilities beyond those supported by our specification metalanguage. One obvious omission, support for floating-point and/or rational arithmetic, would be a straightforward programming exercise. Other necessary extensions, however, constitute genuine research problems. Most notably, there appears to be an unavoidable requirement for a mechanism to schedule some attribute evaluations dynamically, detecting dependency cycles at runtime. This need arises, for example, in the named constant declarations of Modula-2. Since the possibility of a cycle is known in advance, and its detection at runtime is to be handled as an error in the analyzed program (not in the language description) such a mechanism is compatible in principle with strong generation-time checking. In Chapter 12, we suggested an approach that is workable from the standpoint of the specification formalism. It is much less clear whether efficient incremental evaluation can still be provided automatically.

At present, our analysis algorithms handle relations in a simplistic and superficial way. We have suggested several possible improvements, some of which are principally a matter of additional engineering effort and introduce no new conceptual issues. Lurking below the surface, however, is the fact that for relations, much more so than for objects or functions, there is great variability in the kind of implementation that is appropriate in a given context. Indeed, our current, rather naive, implementation of relations is most likely the best one for relations expected to contain only a handful of tuples. A generally satisfactory solution will probably require the provision of many implementation variants, with additional pragmas to allow selection of an appropriate one.

The evaluators generated by Colander II suffer from the naivete of our compiler. Much improvement could be realized from traditional compiler optimizations. Common subexpression elimination, for example, would avoid the overhead of creating unneeded dynamic dependency links during the redundant computations. Other improvements, such as sharing non-local dependency lists among multiple object components, would require new generation-time analyses of the attribute grammar, or yet further guidance from the user in the form of additional pragmas.

In general, our implementation suffers from its attempt to implement the underlying semantics of the specification metalanguage in an incremental fashion, rather than exploiting higher-level understanding of the intended semantics of the analysis. This information is undecidable in general and known only to the user. The problem would be mitigated somewhat if higher-level "chunks" of functionality, such as symbol table management, could be encapsulated in widely re-usable modules or class definitions. A library of hand-optimized implementations could then be provided if needed.

Separate-compilation in the context of attribute grammars remains problematical, as attribute dependency analysis is inherently global. Some allowances for separate compilation were made at the early stages of the Colander II design, but the idea was eventually abandoned completely. It would be useful to introduce some kind of module facility, even if based on simple textual inclusion.

Fibering, while effective, has proven awkward to use in our implementation. A straightforward implementation leads to a proliferation of additional attributes whose significance is not immediately clear, but which clutter the diagnostic reports produced by the dependency analyzer. Localizing the true source of a circularity from the voluminous diagnostics is then difficult. A browser allowing exploration of a graphical display of the attribute dependencies would be of great assistance.

Attribute grammars encourage the user to write the analysis description without thinking through the issue of potential circularities, relying on the the dependency analyzer to discover them. Ultimately, however, the user must develop an understanding of the source of the circularities in order to place the `delayed` pragmas intelligently. Once located, the removal of a circularity may occasionally require significant reformulation of the language description to put it in fiberable form, which would not have been required had the circularity been anticipated. It appears that the generation-time analysis performed by fibering functions primarily as a guarantor of correctness, and not necessarily as a labor-saving device. This is particularly true when extensive use is made of fibering, as in our analysis description for Modula-2. We speculate that encouraging the user to explicitly declare the expected "pass structure" at the outset, and providing some notation for doing so, might be preferable from the standpoint of reasoning about the dependencies during construction of the analysis description.

Finally, limitations of our fibering algorithm may force more components to be fibered than are actually necessary. The algorithm identifies strongly connected components of the dependency graph, not individual cycles, and thus fibers all components contained therein in order to assure that all cyclic dependency paths are severed. Restricting fibering to those components declared as `delayed` is not sufficient in general to limit fibering to a minimal set of components. A more clever algorithm would attempt to identify such a minimal set itself, or might ask for user assistance when confronted with a non-obvious choice.

13.3 Directions for the Future

The most obvious direction for future work would be to directly address these open issues, resulting in a second generation system of the same general character as Colander II. Some issues, such as support for dynamic scheduling and improved fibering, are genuine research topics. Others are just a matter of straightforward engineering or the application of well-known compiler optimization techniques.

We are not convinced, however, that this would be necessarily the best way in which to proceed, particularly if the ultimate aim of the research is to experiment with advanced environments as completed artifacts rather than to further perfect the art of constructing them. Colander II is already a sizeable system reflecting substantial implementation effort, and further development would only make it larger. Furthermore, the attempt to embrace the entire task of incremental analysis in a tidy unified framework necessarily brings with it a degree of brittleness, i.e. the tendency to "hit the wall" when the demands of the task push beyond the natural limits of the tool. In the construction of conventional compilers, interoperating suites of relatively lightweight tools, each focused on a narrow subtask, have been much more widely accepted than monolithic, all-encompassing frameworks. Such tools adapt more easily to awkward real-world requirements, and can usually accommodate *ad hoc* workarounds for situations that do not mesh precisely with the theory upon which the tools are based. The applicability of our incremental evaluation methods would be broadened if they could be decoupled from their present embodiment in a monolithic analyzer generator. Toward this end, we note that the essence of our approach to incrementality is a caching strategy tuned for a certain class of imperative tree-based computations that happens to be appropriate for the programming of static semantic analyzers. In principle, this strategy could be applied to a hand-written analyzer, particularly if caching was used very selectively, as we recommend. It may also be possible to construct a lightweight tool, for example, a preprocessor, that provided automatic assistance in the process.

Just as our approach to incremental analysis is not essentially dependent on attribute grammars, neither are our improvements to attribute grammars in any way dependent on incremental evaluation. Fibering would be a valuable addition to almost any attribute grammar implementation.

The design of Colander II, like its predecessor, is resolutely based on the traditional "textbook" compiler architecture in which parsing is performed prior to semantic analysis and completely independently of it. Many widely used programming languages of practical importance cannot be accommodated in this framework, requiring feedback from semantic analysis to the parser in order to implement a context-sensitive parse. In this case, the usual approach is to interleave the execution of the parser and the semantic analyzer. Subject to restrictions on the acceptable attribute dependencies, it is possible to evaluate an attribute grammar while the AST is being constructed. Colander II could be profitably extended to employ such methods.

As computer hardware continues to improve, the importance of incremental analysis within a single module or source file is greatly diminished, at least from the standpoint of analysis latency. Rising user expectations are resulting in ever-increasing program sizes, however, presenting us with new problems of program scale even as faster hardware sweeps away some of the old ones. In practice, larger programs do not contain larger compilation units, rather they contain more of them. We suggest that the greatest payoff from incremental analysis methods will be realized when they work effectively at the level of an entire system, including those with very large numbers of modules. The persistent state in this case will most likely have to reside on external storage, and be shared by multiple users, that is, it will be a true database in the sense that that term is commonly understood. Developing suitable methods for fine-grained incremental analysis within this context remains an open challenge which has scarcely been addressed.

Bibliography

[1] Harvey Abramson. Definite clause translation grammars. In *International Symposium on Logic Programming*, pages 233–240. IEEE Press, 1984.

[2] Rolf Bahlke and Gregor Snelting. The PSG system: From formal language definitions to interactive programming environments. *ACM Transactions on Programming Languages and Systems*, 8(4):547–576, October 1986.

[3] R. A. Ballance, S. L. Graham, and M. L. Van De Vanter. The Pan language-based editing system. *ACM Transactions on Software Engineering and Methodology*, 1(1):95–127, 1992.

[4] Robert Alan Ballance. Syntactic and semantic checking in language-based editing systems. Technical Report UCB/CSD 89/548, Computer Science Division, University of California, Berkeley, 1989. Ph.D. dissertation.

[5] François Bancilhon and Peter Buneman, editors. *Advances in Database Programming Languages.* ACM Press, 1990.

[6] George McArthur Beshers and Roy Harold Campbell. Maintained and constructor attributes. In *ACM SIGPLAN '85 Symposium on Language Issues in Programming Environments*, pages 34–42, 1985.

[7] Gunther Blaschek and Gustave Pomberger. *Introduction to Programming with Modula-2.* Springer-Verlag, 1990.

[8] P Borras, D. Clément, Th. Despeyroux, J. Incerpi, G. Kahn, B. Lang, and V. Pascual. CENTAUR: The system. In Peter Henderson, editor, *ACM SIGSOFT '88: Third Symposium on Software Development Environments*, pages 14–24, 1988.

[9] John Boyland. Personal communication.

[10] John Tang Boyland. Descriptional composition of compiler components. Technical Report UCB/CSD-96-916, Computer Science Division, University of California, Berkeley, September 1996. Ph.D. dissertation.

[11] Yih-Farn Chen, Michael Y. Nishimoto, and C. V. Ramamoorthy. The C information abstraction system. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[12] Henning Christiansen. Structure sharing in attribute grammars. In J. Małuszyński, editor, *Programming Language Implementation and Logic Programming: International Workshop PLILP '88*, volume 348 of *Lecture Notes in Computer Science*, pages 180–200. Springer-Verlag, 1988.

[13] Keith L. Clark. Negation as failure. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 293–322. Plenum Press, 1978.

[14] B. J. Cornelius. Problems with the language Modula-2. *Software – Practice and Experience*, 18(6):529–543, June 1988.

[15] Declarative Systems, Inc., Palo Alto, CA. *Linguist User's Manual, Version 6.3*, March 1990.

[16] DeGroot and Lindstrom, editors. *Logic Programming: Functions, Equations, and Relations*. Prentice-Hall, 1986.

[17] Pierre Deransart, Martin Jourdan, and Bernard Lorho. *Attribute Grammars: Definitions, Systems and Bibliography*. Springer-Verlag, 1988.

[18] J. Engelfriet. Attribute grammars: Attribute evaluation methods. In B. Lorho, editor, *Methods and Tools for Compiler Construction*, pages 103–138. Cambridge University Press, 1984.

[19] J. Engelfriet and G. Filé. Passes, sweeps, and visits. In S. Even and O. Kariv, editors, *Proceedings of the Eighth International Conference on Automata, Languages, and Programming*, volume 115 of *Lecture Notes in Computer Science*, pages 193–207. Springer-Verlag, 1981.

[20] Joost Engelfriet and Gilberto Filé. Simple multi-visit attribute grammars. *Journal of Computer and System Sciences*, 24:283–314, 1982.

[21] R. Farrow and D. M. Yellin. A comparison of storage optimizations in automatically-generated attribute grammars. *Acta Informatica*, 23(4):393–427, 1986.

[22] Rodney Farrow. Fibered evaluation in Linguist. unpublished draft, Declarative Systems, Inc., Palo Alto, CA.

[23] P. Feiler, S. Dart, and G. Downey. Evaluation of the Rational environment. Technical Report CMU/SEI-88-TR-15, Software Engineering Institute, Carnegie-Mellon University, 1988.

[24] Peter H. Feiler. A language-oriented interactive programming environment based on compilation technology. Technical Report CMU-CS-82-117, Department of Computer Science, Carnegie-Mellon University, 1982. Ph.D. dissertation.

[25] Free Software Foundation, Inc. *Bison Manual*. 59 Temple Place – Suite 330, Boston, MA, 02111-1307. December 1993 Edition for Version 1.23.

[26] Free Software Foundation, Inc. *Flex Manual*. 59 Temple Place – Suite 330, Boston, MA, 02111-1307. Edition 1.03 for Version 2.3.7.

[27] Franz, Inc. *Allegro CL User Guide*, March 1992. version 4.1.

[28] Neal M. Gafter. Parallel incremental compilation. Technical Report 349, Department of Computer Science, University of Rochester, June 1990. Ph.D. dissertation.

[29] Susan L. Graham. Language and document support in software development environments. In *Proceedings of the Darpa'92 Software Technology Conference*, Los Angeles, April 1992.

[30] GrammaTech, Inc. *The Synthesizer Generator^{TM}: Language-Sensitive Editing for CASE*. One Hopkins Place, Ithaca, New York 14850. Undated product overview received in 1993.

[31] Ashish Gupta, Inderpal Singh Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *SIGMOD Record*, 22(2):157–166, June 1993. Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data.

[32] Görel Hedin. *Incremental Semantic Analysis*. Ph.D. dissertation, Department of Computer Science, Lund University, March 1992.

[33] Görel Hedin. An overview of door attribute grammars. In Peter A. Fritzson, editor, *Proceedings of the CC '94 International Conference on Compiler Construction*, volume 786 of *Lecture Notes in Computer Science*, pages 31–51. Springer-Verlag, 1994.

[34] Roger Hoover. Dynamically bypassing copy rule chains in attribute grammars. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languges*, pages 14–25, January 1986.

[35] Roger Hoover. Efficient incremental evaluation of aggregate values in attribute grammars. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 39–50, June 1986.

[36] Roger Hoover. *Incremental Graph Evaluation*. Ph.D. dissertation, Department of Computer Science, Cornell University, 1987.

[37] Roger Hoover. Alphonse: Incremental computation as a programming abstraction. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 261–272, 1992.

[38] Susan Horwitz and Tim Teitelbaum. Generating editing environments based on relations and attributes. *ACM Transactions on Programming Languages and Systems*, 8(4):577–608, October 1986.

[39] Susan B. Horwitz. Generating language-based editors: A relationally-attributed approach. Technical Report TR 85-696, Department of Computer Science, Cornell University, 1985. Ph.D. dissertation.

[40] Paul Hudak et al. Report on the programming language Haskell: A non-strict, purely functional language. *ACM SIGPLAN Notices*, 27(5), May 1992.

[41] Martin Jourdan. An optimal-time recursive evaluator for attribute grammars. In M. Paul and B. Robinet, editors, *International Symposium on Programming: Proceedings of the 6th Colloquium*, volume 167 of *Lecture Notes in Computer Science*, pages 167–178. Springer-Verlag, April 1984.

[42] Martin Jourdan, Carole Le Bellec, and Didier Parigot. The OLGA attribute grammar description lanugage: Design, implementation, and evaluation. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*, pages 222–237. Springer-Verlag, 1990.

[43] Martin Jourdan and Didier Parigot. Internals and externals of the FNC-2 attribute grammar system. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 485–504. Springer-Verlag, 1991.

[44] Catherine Julié and Didier Parigot. Space optimization in the FNC-2 attribute grammar system. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*, pages 29–45. Springer-Verlag, 1990.

[45] Richard K. Jullig and Frank DeRemer. Regular right-part attribute grammars. In *Proceedings of the ACM SIGPLAN '84 Symposium on Compiler Construction*, pages 171–178, 1984.

[46] U. Kastens, B. Hutt, and E. Zimmermann. *GAG: A Practical Compiler Generator*, volume 141 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[47] Uwe Kastens. Ordered attribute grammars. *Acta Informatica*, 13:229–256, 1980.

[48] Uwe Kastens. Attribute grammars as a specification method. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 16–47. Springer-Verlag, 1991.

[49] Uwe Kastens. Implementation of visit-oriented attribute evaluators. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 114–139. Springer-Verlag, 1991.

[50] Peter Lipps, Ulrich Möncke, Matthias Olk, and Reinhard Wilhelm. Attribute (re)evaluation in OPTRAN. *Acta Informatica*, 26, 1988.

[51] Lucid, Inc. Sales literature.

[52] Raul Medina-Mora. Syntax directed editing: Towards integrated programming environments. Technical Report CMU-CS-81-113, Department of Computer Science, Carnegie-Mellon University, March 1982. Ph.D. dissertation.

[53] Josephine Micallef and Gail E. Kaiser. Extending attribute grammars to support programming-in-the-large. *ACM Transactions on Programming Languages and Systems*, 16(5):1572–1612, September 1994.

[54] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML*. MIT Press, 1990.

[55] Robert L. Nord and Frank Pfenning. The Ergo attribute system. In *Proceedings of the Third ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 110–120, 1988.

[56] Joan Peckham and Fred Maryanski. Semantic data models. *Computing Surveys*, 20(3):153–189, September 1988.

[57] Stephen B. Peckham. Globally partitionable attribute grammars. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*, pages 327–342. Springer-Verlag, 1990.

[58] Stephen B. Peckham. Incremental attribute evaluation and multiple subtree replacements. Technical Report 90-1093, Department of Computer Science, Cornell University, Februrary 1990. Ph.D. dissertation.

[59] Maarten Pennings. *Generating Incremental Attribute Evaluators*. Ph.D. dissertation, Utrecht University, 1994.

[60] PROCASE Corporation, 2694 Orchard Parkway, San Jose, California 95134. *SMARTsystem^{TM} Reference Guide*, release 2.0 edition, March 1993.

[61] William Pugh and Tim Teitelbaum. Incremental computation via function caching. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 315–328. ACM Press, January 1989.

[62] Uday S. Reddy. On the relationship between logic and functional languages. In DeGroot and Lindstrom, editors, *Logic Programming: Functions, Equations, and Relations*, pages 3–36. Prentice-Hall, 1986.

[63] Raymond Reiter. On closed world data bases. In Hervé Gallaire and Jack Minker, editors, *Logic and Data Bases*, pages 55–76. Plenum Press, 1978.

[64] T. Reps, C. Marceau, and T. Teitelbaum. Remote attribute updating for language-based editors. In *Proceedings of the Thirteenth Annual ACM Symposium on Principles of Programming Languges*, pages 1–13, January 1986.

[65] Thomas Reps. Generating language-based environments. Technical Report TR 82-514, Department of Computer Science, Cornell University, 1982. Ph.D. dissertation.

[66] Thomas Reps, Tim Teitelbaum, and Alan Demers. Incremental context-dependent analysis for language-based editors. *ACM Transactions on Programming Languages and Systems*, 5(3):449–477, July 1983.

[67] Thomas W. Reps and Tim Teitelbaum. *The Synthesizer Generator: A System for Constructing Language-Based Editors*. Springer-Verlag, 1989.

[68] Graham Ross. Integral C – A practical environment for C programming. In *Proceedings of the Second ACM SIGSOFT/SIGPLAN Symposium on Practical Software Development Environments*, pages 42–48, 1986.

[69] Srinivas R. Sataluri and Arthur C. Fleck. Semantic specifications using logic programs. In Ewing L. Lusk and Ross A. Overbeek, editors, *Logic Programming, Proceedings of the North American Conference 1989*, pages 772–791. MIT Press, 1989.

[70] Guy L. Steele. *Common Lisp: The Language*. Digital Press, 1990.

[71] Doaitse Swierstra and Harald Vogt. Higher order attribute grammars. In H. Alblas and B. Melichar, editors, *Attribute Grammars, Applications and Systems*, volume 545 of *Lecture Notes in Computer Science*, pages 256–296. Springer-Verlag, 1991.

[72] T. Teitelbaum and T. Reps. The Cornell Program Synthesizer: A syntax-directed programming environment. *Communications of the ACM*, 24(9):563–573, September 1981.

[73] J. Uhl, S. Drossopoulos, G. Persch, et al. *An Attributed Grammar for the Semantic Analysis of Ada*, volume 139 of *Lecture Notes in Computer Science*. Springer-Verlag, 1982.

[74] Michael Lee Van De Vanter. User interaction in language-based editing systems. Technical Report UCB/CSD 93/726, Computer Science Division (EECS), University of California, Berkeley, December 1992. Ph.D. dissertation.

[75] Harald Vogt, Doaitse Swierstra, and Mattijs Kuiper. Efficient incremental evaluation of higher-order attribute grammars. In J. Małuszyński and M. Wirsing, editors, *Proceedings of PLILP '91*, volume 528 of *Lecture Notes in Computer Science*, pages 231–242. Springer-Verlag, 1991.

[76] Scott Vorthmann and Richard J. LeBlanc. A naming specification language for syntax-directed editors. In *Proceedings of the IEEE 1988 International Conference on Computer Languages*, pages 250–257, 1988.

[77] Tim A. Wagner and Susan L. Graham. Integrating incremental analysis with version management. In Wilhelm Schafer and Pere Botella, editors, *Software Engineering – ESEC '95*, number 989 in Lecture Notes in Computer Science. Springer-Verlag, 1995. Proceedings of the 5th European Software Engineering Conference.

[78] W. M. Waite. Use of attribute grammars in compiler construction. In P. Deransart and M. Jourdan, editors, *Attribute Grammars and their Applications*, volume 461 of *Lecture Notes in Computer Science*, pages 255–265. Springer-Verlag, 1990.

[79] R. C. Waters. Program editors should not abandon text oriented commands. *SIGPLAN Notices*, 17(7):39–46, 1982.

[80] David A. Watt. An extended attribute grammar for Pascal. *SIGPLAN Notices*, 14(2):60–74, 1979.

[81] David A. Watt and Ole Lehrmann Madsen. Extended attribute grammars. *The Computer Journal*, 26(2):142–153, 1983.

[82] Tim Wilkinson. Kaffe: A free virtual machine to run Java^{TM} code. Web page. `http://www.tjwassoc.demon.co.uk/kaffe/kaffe.htm`.

[83] Taiichi Yuasa and Masami Hagiya. *Kyoto Common Lisp Report.* Research Institute for Mathematical Sciences, Kyoto University, undated, circa 1984. Distributed with the KCL implementation, available at `ftp://ftp.cli.com/pub/kcl/`. See also GNU Common Lisp, at `ftp://ftp.cli.com/pub/gcl/`.

Appendix A

The Syntax of ADL

A.1 Program Units

program_unit → *schema*

program_unit → *body*

schema →
    `language` *identifier* `is`
      { *declaration* }∗
    `end` *identifier* ;

body →
    `language body` *identifier* : *schema_name*
    `is`
      { *definition* }∗
    `end` *identifier* ;

schema_name → *identifier*

A.2 Declarations and Definitions

declaration → `from` *schema_name* `import` *import_list* ;

declaration → `lexeme` *phylum_name* ;

declaration →
    `phylum` *phylum_name*
    [`with`
      { *attribute_declaration* }+]
    `end` *phylum_name* ;

declaration → `phylum` *phylum_name* ;

declaration →
    `operator` *operator_name* `:` *phylum_name* `is`
        { *syntax_form* }∗
    `end` *operator_name* `;`

declaration → `operator` *operator_name* `:` *phylum_name* `;`

declaration → *attribute_declaration*

declaration → `type` *identifier* `;`

declaration → `type` *identifier* `=` *type* `;`

declaration → `datatype` *identifier* `is` { *data_constructor* `|` }+ `;`

declaration →
    `class` *identifier* [`isa` *class_name*]
    [`requiring`
        { *attribute_declaration* }∗]
    [`with`
        { *attribute_declaration* }+]
    `end` *identifier* `;`

declaration → `class` *identifier* [`isa` *class_name*] `;`

definition → `from` *body_name* `:` *schema_name* `import` *import_list* `;`

definition → `lexeme` *phylum_name* `;`

definition →
    `phylum` *phylum_name*
    [`with`
        { *attribute_declaration* }+]
    [`where`
        { *constraint* }+]
    `end` *phylum_name* `;`

definition → `phylum` *phylum_name* `;`

definition →
    `operator` *operator_name* `:` *phylum_name* `is`
        { *syntax_form* }∗
    [`where`
        { *constraint* }+]
    `end` *operator_name* `;`

definition → `operator` *operator_name* `:` *phylum_name* `;`

definition → *attribute_definition*

definition → `collection` *attribute_name* *formal_arguments* [*pragmas*] `;`

definition → `type` *identifier* `=` *type* `;`

definition → `datatype` *identifier* `is` { *data_constructor* `|` }+ `;`

definition →
  `class` *identifier* [`isa` *class_name*]
  [`requiring`
    { *attribute_declaration* }*]
  [`with`
    { *attribute_definition* }+]
  [`where`
    { *constraint* }+]
  `end` *identifier* `;`

definition → `class` *identifier* [`isa` *class_name*] `;`

definition → *constraint*

import_list → `all`

import_list → { *import_item* `,` }+

import_item → *identifier*

phylum_name → *identifier*

operator_name → *identifier*

syntax_form → *identifier* `:` *phylum_name*

syntax_form → `{` *identifier* `:` *phylum_name* `}`

syntax_form → *string_literal*

syntax_form → `{` *identifier* `:` *phylum_name* `}` `*`

syntax_form → `{` *identifier* `:` *phylum_name* `}` `+`

syntax_form → `{` *identifier* `:` *phylum_name* `}` `++`

syntax_form → `{` *identifier* `:` *phylum_name* *string_literal* `}` `*`

syntax_form → `{` *identifier* `:` *phylum_name* *string_literal* `}` `+`

syntax_form → `{` *identifier* `:` *phylum_name* *string_literal* `}` `++`

data_constructor → *identifier*

data_constructor → *identifier* `(` { *element_type* `,` }+ `)`

element_type → *type*

A.3 Attributes

attribute_declaration → `attribute` *attribute_name* `:` *type* `;`

attribute_declaration → `relation` *attribute_name* *formal_arguments* `;`

attribute_declaration → `function` *attribute_name* *formal_arguments* `->` *type* `;`

attribute_definition → `attribute` *attribute_name* `:` *type* [`=` *expression*] [*pragmas*] `;`

attribute_definition → `context` *attribute_name* `:` *type* [*pragmas*] `;`

attribute_definition → `relation` *attribute_name* *formal_arguments* [*pragmas*] `;`

attribute_definition → `function` *attribute_name* *formal_arguments* `->` *type* [*pragmas*] `;`

formal_arguments → `(` { *type* `,` }+ `)`

A.4 Constraints

constraint → *attribute_definition;*

constraint → *attribute_reference* `=` *expression* [`:-` *goal*] `;`

constraint → *attribute_reference* `(` { *pattern* `,` }+ `)` [`:-` *goal*] `;`

constraint → *attribute_reference* `(` { *pattern* `,` }+ `)` `=>` *expr* [`:-` *goal*] `;`

constraint →
    `object` *identifier* `:` *class_name*
    [`where`
      { *constraint* }+]
    `end` *identifier* `;`

constraint → `object` *identifier* `:` *class_name* `;`

constraint →
    `analyze` *operator_child*
    [`with`
      { *attribute_definition* }+]
    [`when [ ] =>`
      { *constraint* }+]
    [`when [` *singleton_name* `] =>`
      { *constraint* }+]

[when [*left_subseq_name* ^ *right_subseq_name*] =>
{ *constraint* }+]
end ;

left_subseq_name → *identifier*

right_subseq_name → *identifier*

constraint → anchor ;

constraint → implement *attribute_reference* as *pragmas* ;

A.5 Naming and Reference

variable → *identifier*

type_name → *identifier*

class_name → *type_name*

constructor_name → *identifier*

node_reference → *identifier*

attribute_reference → *attribute_name*

attribute_reference → *node_reference* . *attribute_name*

attribute_reference → *object_name* . *attribute_name*

component_selection → *expression* . *attribute_name*

module_name → *identifier*

component_name → *identifier*

attribute_name → *identifier*

object_name → *identifier*

A.6 Types

type → *type_name*

type → *type_constructor*

type_constructor → `[` *element_type* `]`

type_constructor → `(` { *element_type* `,` }++ `)`

A.7 Goals

goal → { *literal* `&` }+

literal → *predicate_name* `(` { *pattern* `,` }+ `)`

literal → *expression*

literal → *expression* `=>` *pattern*

literal → ∼ *literal*

literal → `{` *goal* `}`

literal → *expression* `isa` *class_name*

predicate_name → *attribute_reference*

predicate_name → *component_selection*

A.8 Expressions

expression → *numeric_literal*

expression → *string_literal*

expression → `TRUE`

expression → `otherwise`

expression → `always`

expression → `FALSE`

expression → `never`

expression → *variable*

expression → *node_reference*

expression → *attribute_reference*

$\mathit{expression} \rightarrow \mathit{component\_selection}$

$\mathit{expression} \rightarrow \texttt{(}\ \mathit{expression}\ \texttt{)}$

$\mathit{expression} \rightarrow \mathit{attribute\_reference}\ \texttt{(}\ \{\ \mathit{expression}\ \texttt{,}\ \}\texttt{+}\ \texttt{)}$

$\mathit{expression} \rightarrow \mathit{component\_selection}\ \texttt{(}\ \{\ \mathit{expression}\ \texttt{,}\ \}\texttt{+}\ \texttt{)}$

$\mathit{expression} \rightarrow \texttt{[ ]}$

$\mathit{expression} \rightarrow \texttt{[}\ \{\ \mathit{expression}\ \texttt{,}\ \}\texttt{+}\ \texttt{]}$

$\mathit{expression} \rightarrow \texttt{[}\ \{\ \mathit{expression}\ \texttt{,}\ \}\texttt{+}\ \texttt{|}\ \mathit{expression}\ \texttt{]}$

$\mathit{expression} \rightarrow \texttt{(}\ \{\ \mathit{expression}\ \texttt{,}\ \}\texttt{++}\ \texttt{)}$

$\mathit{expression} \rightarrow \mathit{constructor\_name}$

$\mathit{expression} \rightarrow \mathit{constructor\_name}\ \texttt{(}\ \{\ \mathit{expression}\ \texttt{,}\ \}\texttt{+}\ \texttt{)}$

$\mathit{expression} \rightarrow \texttt{not}\ \mathit{expression}$

$\mathit{expression} \rightarrow \mathit{expression}\ \texttt{and}\ \mathit{expression}$

$\mathit{expression} \rightarrow \mathit{expression}\ \texttt{or}\ \mathit{expression}$

$\mathit{expression} \rightarrow \texttt{+}\ \mathit{expression}$

$\mathit{expression} \rightarrow \texttt{-}\ \mathit{expression}$

$\mathit{expression} \rightarrow \mathit{expression}\ \texttt{+}\ \mathit{expression}$

$\mathit{expression} \rightarrow \mathit{expression}\ \texttt{-}\ \mathit{expression}$

$\mathit{expression} \rightarrow \mathit{expression}\ \texttt{*}\ \mathit{expression}$

$\mathit{expression} \rightarrow \mathit{expression}\ \texttt{/}\ \mathit{expression}$

$\mathit{expression} \rightarrow \mathit{expression}\ \texttt{rem}\ \mathit{expression}$

$\mathit{expression} \rightarrow \mathit{expression}\ \texttt{=}\ \mathit{expression}$

$\mathit{expression} \rightarrow \mathit{expression}\ \texttt{/=}\ \mathit{expression}$

$\mathit{expression} \rightarrow \mathit{expression}\ \texttt{<}\ \mathit{expression}$

$\mathit{expression} \rightarrow \mathit{expression}\ \texttt{=<}\ \mathit{expression}$

$\mathit{expression} \rightarrow \mathit{expression}\ \texttt{>}\ \mathit{expression}$

$\mathit{expression} \rightarrow \mathit{expression}\ \texttt{>=}\ \mathit{expression}$

expression → *expression* ^ *expression*

expression → `format` (*string_literal*)

expression → `format` (*string_literal* , { *expression* , }+)

A.9 Patterns

pattern → _

pattern → *variable*

pattern → *expression*

pattern → [ ]

pattern → [{ *pattern* , }+]

pattern → [{ *pattern* , }+ | *pattern*]

pattern → ({ *pattern* , }++)

pattern → *constructor_name*

pattern → *constructor_name* ({ *pattern* , }+)

A.10 Pragmas

pragmas → { *pragma* , }+

pragma → `maintained`

pragma → `ephemeral`

pragma → `delayed` [*identifier*]

Appendix B

An ADL Description for "Example"

B.1 The Schema

```
language Example is

  relation Error(Node, String);

  relation UseOf(Node, Node);

end Example;
```

B.2 The Body

```
language body Example : Example is

  from StringOps : StringOps import all;

  collection Error(Node, String);

  collection UseOf(Node, Node);
```

```
%% String to Integer conversion.

attribute CharZero : Integer = StrChar("0", 0);

function StrToInt(String) -> Integer;
  StrToInt(Str) => StrToIntAux(Str, StrLen(Str)-1);

function StrToIntAux(String, Integer) -> Integer;
  StrToIntAux(Str, Idx) => StrChar(Str, 0) - CharZero
      :- Idx = 0;
  StrToIntAux(Str, Idx) => RestVal * 10 + StrChar(Str, Idx) - CharZero
      :- StrToIntAux(Str, Idx-1) => RestVal;

%% Representation of declared entities.

class BindingStatus
requiring
  % nothing
end BindingStatus;

object Unknown : BindingStatus;
object Undeclared : BindingStatus;

class Entity isa BindingStatus
requiring
  attribute  DeclNode : Node;
end Entity;

class VarEntity isa Entity
requiring
  attribute  Type : TypeShape
      delayed;
end VarEntity;

class TypeEntity isa Entity
requiring
  attribute  Type : TypeShape
      delayed;
end TypeEntity;

%% Representation of types.

datatype TypeShape is TsUNKNOWN
                    | TsINTEGER
                    | TsPOINTER(TypeShape)
                    | TsARRAY(Integer, TypeShape)
                    | TsTYPENAME(TypeEntity)
                    ;
```

```
function EquivTypes(TypeShape, TypeShape) -> Boolean;
  EquivTypes(TsINTEGER, TsINTEGER) => TRUE;
  EquivTypes(TsTYPENAME(Ent), TsTYPENAME(Ent)) => TRUE;
  EquivTypes(TsPOINTER(RTy1), TsPOINTER(RTy2)) => TRUE
      :- EquivTypes(RTy1, RTy2);
  EquivTypes(TsARRAY(Sz1, ETy1), TsARRAY(Sz2, ETy2)) => TRUE
      :- Sz1 = Sz2 &
         EquivTypes(ETy1, ETy2);
  EquivTypes(TsUNKNOWN, Ty) => TRUE;
  EquivTypes(Ty, TsUNKNOWN) => TRUE;
  EquivTypes(_, _) => FALSE;

type TrailType = [Entity];

function TrailMember(Entity, TrailType) -> Boolean;
  TrailMember(Ent, []) => FALSE;
  TrailMember(Ent, [Ent|_]) => TRUE;
  TrailMember(Ent, [_|Rest]) => TrailMember(Ent, Rest);

function CyclicType(TypeEntity) -> Boolean;
  CyclicType(Ent) => CyclicTypeAux(Ent.Type, [Ent]);

function CyclicTypeAux(TypeShape, TrailType) -> Boolean;
  CyclicTypeAux(TsARRAY(_, ETy), Trail)
      => CyclicTypeAux(ETy, Trail);
  CyclicTypeAux(TsTYPENAME(Ent), Trail)
      => TRUE
      :- TrailMember(Ent, Trail);
  CyclicTypeAux(TsTYPENAME(Ent), Trail)
      => CyclicTypeAux(Ent.Type, [Ent|Trail]);
  CyclicTypeAux(_, _)
      => FALSE;

%% Binding environments (symbol table).

class Contour
with
  function  LocalBinding(String) -> BindingStatus;
  function  VisibleBinding(String) -> BindingStatus;
  function  Duplicate(String, Entity) -> Boolean;
end Contour;
```

```
class NullContour isa Contour
requiring
  % nothing
where
  LocalBinding(Ident) => Undeclared;
  VisibleBinding(Ident) => Undeclared;
  Duplicate(Ident, Ent) => FALSE;
end NullContour;

object NullEnv : NullContour ;

class NormalContour isa Contour
requiring
  attribute  Parent : Contour;
  relation   Binds(String, Entity);
where
  implement Binds as maintained;
  implement Duplicate as maintained;
  implement VisibleBinding as maintained;

  Duplicate(Ident, Ent) => TRUE
      :- Binds(Ident, Other) & Other /= Ent;
  Duplicate(Ident, Ent) => FALSE;

  LocalBinding(Ident) => Ent
      :- Binds(Ident, Ent) & ~Duplicate(Ident, Ent);
  LocalBinding(Ident) => Unknown;

  VisibleBinding(Ident) => Ent
      :- LocalBinding(Ident) => Ent & Ent /= Unknown;
  VisibleBinding(Ident) => Ent
      :- Parent.VisibleBinding(Ident) => Ent;
end NormalContour;

type Environment = Contour;

%% Global/predefined environment.

object IntType : TypeEntity
where
  Type = TsINTEGER;
end IntType;

object GlobalEnv : NormalContour
where
  Parent = NullEnv;
  Binds("INTEGER", IntType);
end GlobalEnv;
```

```
%% Phyla.

phylum Program;

lexeme Id;
lexeme IntConst;

phylum Statement
with
  context    Ctx : Environment;
end Statement;

phylum Statements
with
  context    Ctx : Environment;
end Statements;

phylum Declaration
with
  context    Ctx  : Environment;
  relation   Binds(String, Entity);
where
  Binds(Var, Ent) :- never;
end Declaration;

phylum Declarations
with
  context    Ctx  : Environment;
  relation   Binds(String, Entity);
where
  Binds(Var, Ent) :- never;
end Declarations;

phylum TypeSpec
with
  context    Ctx  : Environment;
  attribute  Type : TypeShape;
where
  Type = TsUNKNOWN;
end TypeSpec;

phylum Expression
with
  context    Ctx  : Environment;
  attribute  Type : TypeShape;
where
  Type = TsUNKNOWN;
end Expression;
```

```
phylum Variable
with
  context    Ctx  : Environment;
  attribute  Type : TypeShape;
where
  Type = TsUNKNOWN;
end Variable;

%% Expressions.

operator ConstRef : Expression is
  Val:IntConst
where
  ConstRef.Type = TsINTEGER;
end ConstRef;

operator VarRef : Expression is
  Var:Variable
where
  Var.Ctx = VarRef.Ctx;
  VarRef.Type = Var.Type;
end VarRef;

operator Annotate : Expression is
  "(" Expr:Expression ")"
where
  Expr.Ctx = Annotate.Ctx;
  Annotate.Type = Expr.Type;
end Annotate;

operator Addition : Expression is
  Left:Expression "+" Right:Expression
where
  Left.Ctx = Addition.Ctx;
  Right.Ctx = Addition.Ctx;

  Error(Addition, "Integer expression required") :-
    ~EquivTypes(Left.Type, TsINTEGER);
  Error(Addition, "Integer expression required") :-
    ~EquivTypes(Right.Type, TsINTEGER);

  Addition.Type = TsINTEGER;
end Addition;
```

```
%% Variables

operator SimpleVar : Variable is
  Name:Id
where
  attribute Ent : BindingStatus =
      SimpleVar.Ctx.VisibleBinding(Name.Text);

  Error(Name, "Undeclared variable") :- Ent = Undeclared;

  UseOf(Name, Ent.DeclNode) :- Ent isa Entity;

  SimpleVar.Type = Ent.Type :-
      Ent isa VarEntity;
  SimpleVar.Type = TsUNKNOWN :-
      otherwise;
end SimpleVar;

operator SubscriptedVar : Variable is
  Var:Variable "[" Idx:Expression "]"
where
  Var.Ctx = SubscriptedVar.Ctx;
  Idx.Ctx = SubscriptedVar.Ctx;

  Error(Var, "Subscripted variable must be an array") :-
      ~Var.Type => TsUNKNOWN &
      ~Var.Type => TsARRAY(_, _);

  Error(Idx, "Index must be an integer expression") :-
      ~Idx.Type => TsUNKNOWN &
      ~Idx.Type => TsINTEGER;

  SubscriptedVar.Type = EltTy :-
      Var.Type => TsARRAY(_, EltTy);
  SubscriptedVar.Type = TsUNKNOWN :-
      otherwise;
end SubscriptedVar;
```

```
operator DereferencedVar : Variable is
  Var:Variable "^"
where
  Var.Ctx = DereferencedVar.Ctx;

  Error(Var, "Dereferenced variable must be a pointer") :-
      ~Var.Type => TsUNKNOWN &
      ~Var.Type => TsPOINTER(_);

  DereferencedVar.Type = RefTy :-
      Var.Type => TsPOINTER(RefTy);
  DereferencedVar.Type = TsUNKNOWN :-
      otherwise;
end DereferencedVar;

%% Type specifiers.

operator NamedTypeSpec : TypeSpec is
  TypeName:Id
where
  attribute Ent : BindingStatus =
      NamedTypeSpec.Ctx.VisibleBinding(TypeName.Text);

  Error(TypeName, "Undeclared type name") :- Ent = Undeclared;

  UseOf(TypeName, Ent.DeclNode) :- Ent isa Entity;

  NamedTypeSpec.Type = TsTYPENAME(Ent) :-
      Ent isa TypeEntity;
  NamedTypeSpec.Type = TsUNKNOWN :-
      otherwise;
end NamedTypeSpec;

operator ArrayTypeSpec : TypeSpec is
  "array" "[" Size:IntConst "]" "of" EltTy:TypeSpec
where
  EltTy.Ctx = ArrayTypeSpec.Ctx;

  ArrayTypeSpec.Type = TsARRAY(StrToInt(Size.Text), EltTy.Type);
end ArrayTypeSpec;

operator PointerTypeSpec : TypeSpec is
  "pointer" "to" RefTy:TypeSpec
where
  RefTy.Ctx = PointerTypeSpec.Ctx;

  PointerTypeSpec.Type = TsPOINTER(RefTy.Type);
end PointerTypeSpec;
```

```
%% Declarations.

operator TypeDecl : Declaration is
   "type" Name:Id "=" Ty:TypeSpec
where
  Ty.Ctx = TypeDecl.Ctx;

  object TypeObj : TypeEntity
  where
    Type = Ty.Type;
    DeclNode = Name;
  end TypeObj;

  TypeDecl.Binds(Name.Text, TypeObj);

  Error(Name, "Cyclic type definition") :-
      CyclicType(TypeObj);

  Error(Name, "Multiply-declared identifier") :-
      TypeDecl.Ctx.Duplicate(Name.Text, TypeObj);
end TypeDecl;

operator VarDecl : Declaration is
   "var" Var:Id ":" Ty:TypeSpec
where
  Ty.Ctx = VarDecl.Ctx;

  object VarObj : VarEntity
  where
    Type = Ty.Type;
    DeclNode = Var;
  end VarObj;

  VarDecl.Binds(Var.Text, VarObj);

  Error(Var, "Multiply-declared identifier") :-
      VarDecl.Ctx.Duplicate(Var.Text, VarObj);
end VarDecl;
```

```
%% Declaration sequences.

operator DeclList : Declarations is
  { Decls:Declaration ";" }*
where
  Decls.Ctx = DeclList.Ctx;

  analyze Decls
  with
    context    Ctx  : Environment;
    relation   Binds(String, Entity);
  when [] =>
    % empty sequence
    Decls.Binds(Ident, Ent) :- never;
  when [ d ] =>
    % singleton sequence
    anchor;
    d.Ctx = Decls.Ctx;
    Decls.Binds(Ident, Ent) :- d.Binds(Ident, Ent);
  when [ d1 ^ d2 ] =>
    % nondeterministic sequence split
    anchor;
    d1.Ctx = Decls.Ctx;
    d2.Ctx = Decls.Ctx;
    Decls.Binds(Ident, Ent) :- d1.Binds(Ident, Ent);
    Decls.Binds(Ident, Ent) :- d2.Binds(Ident, Ent);
  end;

  DeclList.Binds(Ident, Ent) :- Decls.Binds(Ident, Ent);
end DeclList;

%% Statements.

operator Assignment : Statement is
  Var:Variable ":=" Val:Expression
where
  Var.Ctx = Assignment.Ctx;
  Val.Ctx = Assignment.Ctx;

  Error(Assignment, "Incompatible types in assignment") :-
      ~EquivTypes(Var.Type, Val.Type);
end Assignment;
```

```
operator Block : Statement is
  "declare"
      { Decls:Declarations }
  "begin"
      { Stmts:Statements }
  "end"
where
  Decls.Ctx = BodyCtx;

  object BodyCtx : NormalContour
  where
    Parent = Block.Ctx;
    Binds(Ident, Ent) :- Decls.Binds(Ident, Ent);
  end BodyCtx;

  Stmts.Ctx = BodyCtx;
end Block;

%% Statement sequences.

operator StmtList : Statements is
  { Stmts:Statement ";" }*
where
  Stmts.Ctx = StmtList.Ctx;

  analyze Stmts
  with
    context    Ctx : Environment;
  when [] =>
    % empty sequence
  when [ s ] =>
    % singleton sequence
    anchor;
    s.Ctx = Stmts.Ctx;
  when [ s1 ^ s2 ] =>
    % nondeterministic sequence split
    anchor;
    s1.Ctx = Stmts.Ctx;
    s2.Ctx = Stmts.Ctx;
  end;
end StmtList;
```

```
  %% Program.

  operator Prog : Program is
    { Body:Statements }
  where
    Body.Ctx = GlobalEnv;
  end Prog;

end Example;
```

Appendix C

The Modula-2 Language Description

The Modula-2 language description has been omitted from this technical report due to space considerations. It may be obtained electronically from `http://sunsite.berkeley.edu/NCSTRL`.