University of Washington

Abstract

Program Restructuring as an Aid to Software Maintenance

by William G. Griswold

Chairperson of the Supervisory Committee: Professor David Notkin
Department of Computer Science
and Engineering

Maintenance tends to degrade the structure of software, ultimately making maintenance more costly. At times, then, it is worthwhile to manipulate the structure of a system to make changes easier. However, it is shown that manual restructuring is an error-prone and expensive activity. By separating structural manipulations from other maintenance activities, the semantics of a system can be held constant by a tool, assuring that no errors are introduced by restructuring. To allow the maintenance team to focus on the aspects of restructuring and maintenance requiring human judgment, a transformation-based tool can be provided—based on a model that exploits preserving data flow-dependence and control flow-dependence—to automate the repetitive, error-prone, and computationally demanding aspects of restructuring. A set of automatable transformations is introduced; their impact on structure is described, and their usefulness is demonstrated in examples. A model to aid building meaning-preserving restructuring transformations is described, and its realization in a functioning prototype tool for restructuring Scheme programs is discussed.

Table of Contents

List of Figures

List of Tables

Acknowledgements

I cannot thank enough those who have helped me to reach this point, and it is not possible to thank them all here. But I thank especially Maureen Feeley for always believing in me more than I believed in myself, and never letting me forget. Now it's her turn. I thank my advisor, David Notkin, for his unfailing guidance and friendship beyond mortal cause, and his wife, Cathy Tuttle, for letting him work nights and weekends with me over the networks between the US and Japan. I also thank David and Cathy for knowing me better than I know myself, and their wisdom and generosity in using that knowledge. I thank my friends who helped me in my last year, my fellow possums, Calvin Lin and James Ahrens, who insisted I laugh through all my troubles. I thank Kevin Sullivan for telling me what my thesis was about whenever I forgot.

I thank James Larus for use of his Curare system and his advice on its workings. Thanks, too, to David Garlan, Harold Ossher, and Ron Cytron for various kinds of advice.

I thank my reading committee, Alan Borning and Alan Shaw, for enriching my thesis through their unique perspectives.

I thank Larry Snyder for my first opportunity to do languages research, and to Gail Alverson, Larry, and David for making the collaboration so fruitful and fun.

I thank my father Ralph Griswold, and my step-mother, Madge Griswold, for holding my common sense in escrow while completing my dissertation. I thank my mother, Ann Wood, and my step-father, Belford Wood, for reminding me I never had any common sense, but that I was alright anyway.

I thank my Grandmother, Inez Griswold, for the courage to keep me in her hopes and thoughts this past year. This dissertation is dedicated to my Grandfather, Gale Griswold, 1902–1991, who taught me wonder, how to capture it on film, and, I hope, how to give it to others.

Disk, anyone?

Chapter 1

Introduction

1.1 The Maintenance Problem

Computer software has been touted as a panacea for the engineer because it is so malleable compared to physical construction media such as concrete, steel, and silicon. However, its apparent flexibility has not been successfully exploited—software maintenance (enhancement and repair) remains disproportionately expensive relative to the expected cost of the required changes and the quality of the resulting software.

In studies of OS/360 and other large systems, L.A. Belady and M.M. Lehman observed that the cost of a change grew exponentially with respect to a system's age [Belady & Lehman 76a]. They associated these rising costs with decaying structure caused by the accumulation of unanticipated changes:

> The addition of any function not visualized in the original design will inevitably degenerate structure. Repairs also, will tend to cause deviation from structural regularity since, except under conditions of the strictest control, any repair or patch will be made in the simplest and quickest way. No search will be made for a fix that maintains structural integrity [Belady & Lehman 71, p. 113].

They conclude that this cannot go on indefinitely without having to rebuild the system from scratch, and the need to minimize software cost

> suggests that large-program structure must not only be created but must also be maintained if decay is to be avoided or postponed. Planning and control of the

> maintenance and change process should seek to ensure the most cost-effective balance between functional and structural maintenance over the lifetime of the program. Models, methods and tools are required to facilitate achieving such balance [Lehman 80, p. 383].

The structural principle of information hiding is used for the initial design of a system to isolate each design decision that is likely to change in a module. Effective isolation increases software flexibility by reducing the cost of changing these volatile choices [Parnas 72]. But when an *unanticipated* need for change arises—say, due to demands for enhancement from users—some decisions motivating the original module structure will no longer be valid. This implies that the changes will cross module boundaries, and module interfaces and implementations will have to be modified [Parnas 79]. This need for non-local change is the key property of structural degradation.

One way structural degradation can be addressed is to restructure the program to isolate the design decisions that must evolve in reaction to an unanticipated need for change. Maintaining structure, however, is itself a complex and costly activity. Indeed, it is just like making a functional change in a structurally inadequate system. In particular, global search and change is required to maintain the behavioral relationships between a modified interface and the references to it throughout the program. This, in part, is why there is a preference for quick fixes over those that retain or improve structural integrity.

The thesis of this research is that global restructuring can be cost-effective, but only if automated and separated from other qualitatively different maintenance activities. In particular, it is claimed that:

- The cost of manual restructuring is, in practice, proportional to the complexity of the software.
- Automating restructuring reduces its costs by performing the non-local aspects of the restructuring.
- Automating restructuring avoids introducing costly errors into the system.

To effectively isolate design decisions, automated restructuring must support changing the interface relationships between modules so that what otherwise would be intermodule changes become intramodule changes. Remodularization requires more than automated removal of gotos or control flow restructuring [Böhm & Jacopini 66][Williams & Ossher 77][Morgan 84]. For example, it requires operations such as adding or removing a function from a module interface, changing a function interface, or hiding exposed representation in a module. And since these restructuring operations are intended to ease future maintenance, the software engineer must be able to direct the restructuring towards aiding anticipated future maintenance.

To address these requirements, this thesis proposes automating restructuring so that the software engineer applies a local structural change to a syntactic form in the program, and the tool performs the compensating changes—potentially distributed throughout the program—necessary to prevent introducing inconsistencies into the program. This frees the engineer of the highest costs of structural changes: non-local search and change, and later, debugging to repair inconsistencies in the distributed changes. Furthermore, this approach leaves the engineer in control of the subjective activities of choosing the appropriate structure.

1.2 Possible Solution Approaches

The range of approaches for solving the problems of structural maintenance fall into the categories of models, languages, methods, and tools. The models approach is represented by Belady and Lehman's work, discussed further in Section 1.4.2.

A languages solution entails designing or extending a language to accommodate the evolution of structure in a program. Object-oriented languages are representative of this approach. An object-oriented language permits adding function to a program, if designed properly [Cunnington et al. 90], by incrementally, relatively locally augmenting the existing class structure—modifying existing classes (modules, of a sort) is not necessary. This postpones restructuring for a class of extensions.

A methods solution provides techniques—guidelines—that a software engineer can apply to build a system so that it is easy to evolve, or to restructure it if it has become difficult to evolve. Parnas's guidelines for module [Parnas 72] and virtual machine [Parnas 79] design are examples of techniques that can postpone structural modification of a system by assuring locality of change for likely changes. Methods are largely language independent, although the presence of language mechanisms to support the methods can be helpful. For example, the package construct in Ada helps describe and enforce system decomposition in Ada programs.

Although the other approaches are essential, the inherent costs of restructuring—managing the consistency of non-local changes—cannot be reduced without automation (See Section 1.4). A tools solution provides computer support for structural activities that are difficult for the user to perform correctly or quickly, but are easy for a computer to perform. For maintaining software structure, a tool might ease access to a language's features, help reveal structural properties of a program, or aid in the keeping track of auxiliary information related to system structure and evolution. For example, the Revision Control System [Tichy 82] captures the change history of a program and allows versions to be tracked. If errors are introduced into a tracked program, earlier program versions can be retrieved to return to a working version or help find the change that caused the error. Like a language, a tool can make methods easier to follow or apply. A tool that discovers component relationships [Embley & Woodfield 88][Rich & Waters 88][Schwanke 91][Selby & Basili 91] or that automates remodularizing a program can aid an engineer in using Parnas's system decomposition principles to achieve locality of change.

1.3 A Tools Solution—Restructuring

This thesis focuses on a tools solution to mitigate the software maintenance problem. The goal is to automate the manipulation of program structure to reduce the cost of structural evolution. This approach is based on the premise that modifying a program whose structure is not natural for the given change can be more costly than first restructuring it to a more accommodating form with the aid of a tool, and then modifying it. It also supposes, as Belady and Lehman claim, that structural degradation occurs naturally, and cannot be avoided by forethought or language mechanisms [Belady & Lehman 71].

Changes that cross module boundaries are the ones that will be costly to perform [Parnas 72]. This includes manual restructuring. As an example, consider a pair of routines that must always be called in a particular order, such as an allocation function and an initialization function. Suppose this is unenforced in their implementation, but now it is desired to enforce the ordering by bundling the two calls into a new allocate–and–initialize function. This requires not only exporting the new function and unexporting the other two, but also requires changes at every site where the individual functions are called, potentially in many modules, and thus costly and error-prone.

Applying such non-local updates and assuring their consistency to avoid introducing errors are what make restructuring difficult. A tool can help by managing the distributed updates so that they preserve the desired semantic properties of the program. By automating only this aspect of the restructuring process, the software engineer is freed of the error-prone aspects of restructuring, but is left in control of the subjective task of choosing an appropriate structure for making an enhancement or repair.

A tool can perform the non-local aspects of structural changes such as bundling a pair of calls into a single call, replacing an expression with a variable that has its value, swapping the formal parameters in a procedure's interface and the respective arguments in its calls, adding a parameter to a procedure definition and the appropriate argument to its calls, replacing in-line code with a call to a function that contains that code, and hiding exposed representation in a module, to name a few.

One way a tool can automate a structural change is to take a "local" structural change by the engineer, and then find the distributed updates necessary to preserve the meaning—that is, the correspondence between inputs and outputs—of the program. For example, in the abbreviated program in Figure 1.1, when the engineer swaps `push`'s formal parameters to conform to interface rules, the tool would be responsible for swapping the arguments in its calls. If evaluation order of the arguments might affect the values of the arguments—as in the call of `h(myStack)` in the second call on `push`—the tool would be responsible for prohibiting the change and alerting the engineer to the problem. For a restructuring tool to be precise enough to be useful, then, it must be able to examine the definitions of functions (such as `h`) to determine properties such as side-effects to arguments.

```
procedure push(s, v)
  insert(v, s.head)
  return s
end
      .
      .
      .
push(myStack,1)
      .
      .
      .
push(myStack,h(myStack))
```

$\Rightarrow$

```
procedure push(v, s)
  insert(v, s.head)
  return s
end
      .
      .
      .
push(1,myStack)
      .
      .
      .
push(h(myStack),myStack)
```

Figure 1.1: Swapping the parameters of procedure push

A tool that preserves meaning by making the necessary non-local changes assures making coherent distributed changes. In particular, given an engineer's change, the tool takes responsibility for finding the components to be updated and for preserving the program's meaning. This frees the engineer of the searching and updating necessary to complete a non-local change, as well as checking if the changes are sufficient to preserve meaning. Also, this model enables the engineer to locally specify the global restructuring—the engineer's change to the definition of `push` implies for the tool changes in all its uses.

1.4 Restructuring and the Maintenance Process

Can restructuring reduce the cost of maintenance? A first step towards answering this question is to establish a basis for evaluating the potential success of restructuring. Then a tool can be built on the premises of the model, and experiments conducted to see if the data fits the model. This section develops a maintenance cost model accommodating restructuring.

It is common knowledge that maintenance is the most expensive component of the overall software process, running as high as 70% [Lientz & Swanson 80]. Belady and Lehman have related software structure and the cost of maintenance at the macro level of detail, providing an equation that relates system age, structure, and maintenance costs. Parnas's work is less quantitative, but provides guidelines for building structurally robust programs. Although Parnas's and Belady and Lehman's research do not explicitly address restructuring, their work links a program's structure and the cost of its maintenance. This link is used to argue that tool-aided restructuring can lower that cost.

1.4.1 Modules and Structure

The primary structuring mechanism for reducing the cost of changes is abstraction, particularly through the module and the procedure. A module (or procedure) hides a design decision, allowing changes to that decision without affecting other parts of the program [Parnas 72]. Parnas emphasizes that modules are used to isolate design decisions that the designer *anticipates* are likely to change [Parnas 79]. Accordingly, any change not accounted for by the designers can affect many modules, rather than just one, and thus change their interfaces [Parnas 79], which is costly [Parnas 72]. Thus a module decomposition should establish interfaces between modules that are unlikely to change when changes are made to the program. However, user-driven enhancement implies that enhancements can be proposed that were not anticipated by the original designers.[1] Thus, in spite of the designers' best efforts, unanticipated changes, and hence non-local ones, will arise.

Also, given the range of possible future enhancements, it is not economically feasible to accommodate every one [Boehm 81, pp. 20–21][Parnas 79]; given a set of design decisions to hide, there may be *conflicting* feasible modularizations of a program. One must be chosen in favor of the others.

As an example of conflicting modularizations, suppose a programmer is designing an interpreter for a polymorphic programming language. Following traditional modularization techniques, the programmer factors the program into the types in the language. There is also a module `polymorphic` that knows about how each operation in each type is combined into a polymorphic operation in the language. Now suppose the type `set` (implemented with hash tables) is added to the interpreter. It can be added with local change: a `set` module is added for the set type, and the module `polymorphic` is changed to account for its operations. Now suppose a new operation, such as an iterator for aggregates is added. Then it is necessary to modify every module encapsulating an aggregate type, plus the module `polymorphic`. These two enhancements—along the orthogonal "dimensions" of type versus function—suggest different modularizations; however the designer can choose only one.[2]

The notions of *coupling* and *cohesion* can be used to help manage modularization tradeoffs [Stevens et al. 74][Embley & Woodfield 88]. A relationship between two components in different modules is *coupling.* In contrast, a relationship between two components within a module is called *cohesion.* There are several classes of coupling and cohesion, some desirable, some not, based on their ability to localize the impacts of soft-

[1]Alan Perlis asserts that modifications to software are determined by "traffic", i.e., the kind and quantity of usage a piece of software undergoes [Perlis 88]. Only software that gets used heavily is likely to be changed, and the style of usage determines the kinds of changes requested.

[2]A class-based language can represent the polymorphism in an inheritance graph. However, this distributes the type and function information among superclasses and subclasses. In particular, the change for adding the iterator would be non-local, requiring changes to the aggregate superclass and all its subclasses that do not share representation (i.e., iteration for lists vs. hash tables is different).

ware change. By choosing to structure a system that favors the desirable classes (for those design decisions that are most likely to change), the structure should allow local change during maintenance. Likewise, an engineer can employ these design rules during restructuring [Stankovic 82], but using the newly evolved design goals.

For example, two kinds of coupling are common coupling and data coupling. Common coupling[3] is the relationship due to two modules sharing global data. Data coupling is sharing through parameter passing in procedure calls. The latter kind of sharing is preferred. Two kinds of cohesion are control cohesion and informational cohesion. Two components are control-cohesive if they reside in the same module and are related primarily by the fact that they are referenced in close time-proximity to each other. Control cohesion is considered to be fairly weak, and so not very desirable. Information cohesion is the intramodular relationship between a data representation and the functions that directly access it. Information cohesion—typically manifested in a good data abstraction—is very desirable, especially in contrast to its (potential) coupling counterpart, representation coupling.

However, when the potential for conflicting design choices is combined with the uncertainty of which changes will be requested, it is likely that the choices made will not be optimal, even when managed by coupling-cohesion tradeoffs. The non-local change implied will require a programmer to understand and manipulate a significant proportion of a program's components—often textually dispersed and subtlely related—increasing the chance of introducing errors when performing the change. As an example of the subtlety of component relationships, a study of a software product's maintenance revealed that 53% of the defects introduced by adding new product features were to existing features [Collofello & Buck 87]. Also, removing an error during maintenance is expensive—as expensive as a typical change during maintenance [Boehm 81, pp. 40–41] (See the next paragraph). Unsuitable modularization for a specific enhancement, then, is potentially a high cost of maintenance, and so automated restructuring must support

[3]Common coupling gets its name from the FORTRAN COMMON statement, used to share global data between functions.

remodularization. Of course, during restructuring itself it is necessary to cope with the subtleties in software to avoid expensive errors. This implies that a primary requirement of automated module restructuring is is preventing the introduction of errors during restructuring.

1.4.2 Structure and Complexity

Parnas identifies an unavoidable source of high maintenance cost and describes how it can be reduced, but he makes no predictions about the actual cost of changes. In their studies of OS/360 and other large systems, Belady and Lehman observed that the cost of maintenance grew exponentially with respect to a system's age [Belady & Lehman 76a]. Based on this evidence, Belady and Lehman defined a model of maintenance cost based on a program's complexity at release i [Belady & Lehman 71]:

$$C_i = 2^{2G(i)-DAL(i)}$$

$G(i)$ represents the complexity of the system, which as an observable variable can be quantified as the percentage of modules handled in a release interval [Belady & Lehman 76b]. $DAL(i)$ (known as "Documentation, Accessibility, and Learnability") represents the quality of the documentation and programmer comprehension of the system. By improving documentation and increasing programmer familiarity with a system, complexity can be controlled. Also, since the size of a system tends to grow linearly with respect to the release interval number, the complexity of a system grows exponentially in relation to its size. This model for complexity equates to the cost of a change, since to make a correct change requires cross-checking it for consistency for an exponential number of relationships.

The exponential growth in complexity is due to *stratification* of changes during the system's history [Belady & Lehman 71]. A way to understand this intuitively is to look at the fault structure of the system relative to changes. From the base system an attempt to repair an error will remove one error, and perhaps inject others accidentally.

If removing this error was not the last error, there are residual errors as well. This means that there are now two kinds of errors, old ones and new ones, introduced at different times for different reasons. If this occurs at each error-correction step, then there is a binary tree of corrections, each leaving residual errors and generating new ones, thus growing the tree (See Figure 1.2). The layering of changes yields a network of relationships between the program base and amongst the repairs. In fact there are $2^{G(i)}$ individual strata, and $2^{G(i)}(2^{G(i)}-1)$ pairwise relationships among them to be considered (hence $2^{G(i)} + 2^{G(i)}(2^{G(i)} - 1) = 2^{2G(i)}$ of the complexity equation). This same structure arises in the system whether the change is an error repair or an enhancement, since the enhancement is like a repair that is not removing residual errors. Note also that even if all the errors are removed, the strata persist as the network of *corrections* of the errors. That is, the leaves of the tree are single child residuals with no generated errors.

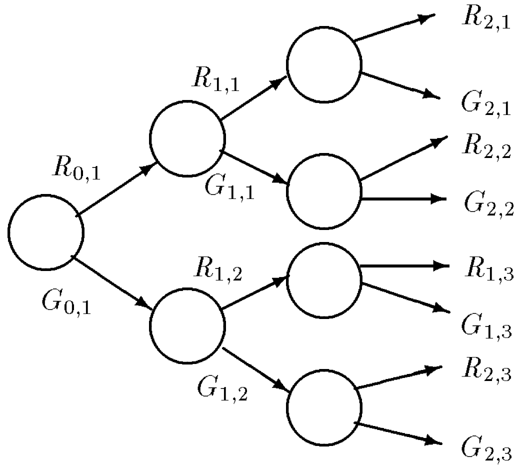


Figure 1.2: Program change strata due to residual (R) and generated (G) errors

Any change that is made by a programmer must be checked for consistency with every element in the strata, requiring exponential work, as well as requiring repair of affected elements [Belady & Lehman 71]. If this work is not done, then the error strata are likely to be expanded, requiring greater work in the future to remove generated errors.

From Parnas's perspective, the increasing number of relationships suggests that the potential for locality of change is decreasing, and that module interfaces are more likely to be changing. In essence, the model quantifies—asymptotically—Parnas's claim that it is impossible to maintain locality of change indefinitely.

Belady and Lehman concur, concluding that exponential growth in complexity is inevitable, and increasing DAL can only delay the effects [Belady & Lehman 71]. They complement their conclusion with the claim that *progressive* activities such as enhancement require continual *anti-regressive* effort to maintain the documentation and programmer familiarity on such changes. However, anti-regressive activities in practice get ignored under financial and time pressures, and because they are not usually as psychologically satisfying as progressive activities. As complexity increases, so does the need for anti-regressive activity. This can be maintained only while complexity is not too large (since the cost of DAL is likely to be related to complexity), at which point system replacement becomes more economical.

The relationship between complexity and structure is simple: the structure of a system is manifested by the components that must be handled (updated or checked) when a change is made. All those components handled for a change are related through that change. The cost of a change is related (exponentially) to the size of the relation. A change is local if a small number of components are handled during a change, but global if a large number of components are handled. Using Parnas's model of structure, small changes do not change module interfaces (the number of components handled equals one), but large changes do (the number of components handled is greater than one). The latter is reflected in the equation of Belady and Lehman as high handle rates (over a release interval) and thus large C. In both models, then, good structure means that changes affect few components, and that module interfaces are not affected.

1.4.3 Injecting Tool-Aided Restructuring into the Model

Given that good structure requires stable interfaces, and also that stable interfaces are improbable given user-driven changes, restructuring is required to create stable module interfaces for a change. However, the model predicts that (manual) restructuring, being undifferentiable from any other kind of change, will be just as costly.

Automated restructuring is different. First, the automation preserves meaning, so no errors are generated, and no strata are added. Second, the knowledge of the program by the restructuring tool is perfect (for example, it can find all uses of a variable—albeit conservatively—whereas an engineer might make errors due to mistracing pointers, missing some files, etc.) and it comes at low cost since it is fully automated. This means, according to the equation above, that $2G(i) = DAL(i)$, and so restructuring has constant cost with respect to release number, and thus constant complexity. Note restructuring cannot reduce the number of existing faults, since meaning is preserved. However, if restructuring is targeted to the change, so that it is localized within a module, the number of strata elements and their pairwise interactions of elements *for the proposed change* is reduced drastically. This would cause a large reduction in complexity for the change.

The Belady and Lehman model does not try to quantify such a situation. In particular, complexity is now being judged relative to a particular change, and restructuring can lower the complexity of a change at little cost. Changing indexing in the complexity equation to reflect changes rather than releases would bring the model closer to Parnas's observation that the cost of a change depends upon it being anticipated. But modifying the indexing of the complexity equation to reflect changes rather than releases is not without problems. For instance, a change interval will typically be much smaller than a release interval, and some activities (say, perhaps, reorganizing a project's personnel organization) can only occur on a per-release basis.

Another variable not modeled accurately is that relationships between modules are viewed as much more expensive to manage then those within a module. If the restructuring tool helps to perfectly localize a change, then G goes to zero [Belady & Lehman 71], and thus the cost of a change goes to zero, which is not precise since it ignores the time using the tool. Also reducing G probably has a negative impact on DAL. However, Belady and Lehman have observed that, asymptotically, G dominates DAL, and in fact that as G increases, DAL decreases. So attacking G even at the expense of DAL is the best choice, and it appears complexity—for the particular change—will be small.

The Belady and Lehman model projects a low cost for automated restructuring or the subsequent change itself, but no measure. Unfortunately, no simple enhancement of the model is possible. The original model provides no quantitative definition for functions G and DAL, so it is impossible to enhance them. Also, the Belady and Lehman model is supported by substantial quantitative data [Belady & Lehman 76a] and cogent theory [Belady & Lehman 71][Belady & Lehman 76a]. The change in indexing, effects on G and DAL, and the actual cost of restructuring must be derived in the same manner. Experience with a tool like the one described in this thesis will help define a G that reflects the benefits of automated restructuring.

The analysis above suggests that restructuring should occur before system structure has degenerated too much, and that automated restructuring can reduce maintenance costs considerably. Restructuring may profitably occur on a per-change basis, since if the estimated cost of a change (without restructuring) is judged to be too high, restructuring will be appropriate if its cost plus the cost of the subsequent (local) change is not be above that of the original estimate. This, of course, does not account for the potential long-term effects of a restructuring, which are harder to measure.

1.5 A Restructuring Example

To give a feeling for the sorts of structural problems encountered during maintenance and how a tool can help, consider a community transit system that actively tracks the distance traveled by each bus on the road. The bus module exports this value as variable `miles-traveled` (See Figure 1.3).[4] Every three minutes the bus sends its trip odometer reading to the central computer, which assigns it to `miles-traveled`. This information is used by the tracking module for displaying the location of the bus on a map for the dispatcher. Also, the total accumulated miles are used for scheduling preventive maintenance of the buses.

When the bus authority decides to put buses onto ferries to service nearby islands, `miles-traveled` cannot be used for both scheduling maintenance and also locating a bus, since not all travel miles will be rolling miles. A new abstraction is needed to separate rolled miles from those due to ferry trips. Further, it is necessary to determine which expressions currently using `miles-traveled` should be just rolled miles, and which should be the combined value.

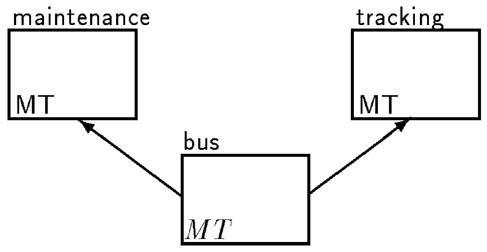


Figure 1.3: module export–import structure of the initial transit system (MT = `miles-traveled`, definition in italics)

The objective of the software engineer is to add ferries to the system without changing the behavior of the system with regard to old (non-ferry) inputs. With a restructuring tool the engineer can locally specify structural changes without changing the meaning of the program. By preceding the enhancement with restructuring, the changes needed to implement the enhancement will be local, and thus easier to perform. Automated

[4]This example is unrealistic, since it handles only one bus. In a more realistic example, the `miles-traveled` variable would probably be a record field of the bus type to allow an arbitrary number of buses. The example would be essentially the same in this form.

restructuring will also reduce the number of changes that can contain new errors.

First, the bus module variable `miles-traveled` needs to be renamed to `rolled-miles` to reflect more precisely its true meaning. With the tool the engineer invokes the transformation `rename-variable(`*miles-traveled declaration*`, "rolled-miles")`, which changes the names of all references of the variable to `rolled-miles`. This is not just a syntactic change, since only references to that name within that scope should be changed. Also, for the name change to preserve meaning, the tool must verify that the new name does not conflict with any names in the modified scope.

Now a new abstraction, `total-miles`, must be created, which will eventually be the combined value of `rolled-miles` and ferry trip miles. Everywhere `rolled-miles` gets the new odometer value, so should `total-miles`. However, it is not satisfactory to represent `total-miles` as a variable, and add the code `total-miles := rolled-miles` where `rolled-miles` is updated. One reason is that it distributes the updates to `total-miles`, and hence later will force global updates to make the ferry miles enhancement. Also, if updates to ferry miles are represented as increments rather than total miles, assigning `rolled-miles` to `total-miles` will overwrite the accumulated ferry miles.

To avoid these problems, the engineer introduces a function `total-miles` that returns `rolled-miles`. This is done by invoking `make-function(`*use of miles-rolled*`, "total-miles")`. This defines a function whose value is `rolled-miles` when called. Then the engineer invokes `global-substitute-function(total-miles)`, which brings each expression equivalent to the function—each reference to `rolled-miles`—to the engineer for approval to substitute. The engineer approves the matches in the bus tracking module and prohibits the ones in the bus maintenance module. Meaning is preserved by these substitutions because each call exactly represents the reference to `rolled-miles` that was replaced.

If the search finds an instance that *updates* `rolled-miles`, and the engineer wants it to be `total-miles`, there is probably an encapsulation violation in the system. That is, `rolled-miles` should be updated solely by readings from the odometer in the bus module. However, this could occur if there is already another kind of mile that is not strictly a rolling mile. The substitution by `global-substitute-function` on such a match is prohibited because it is syntactically illegal to assign a value to a function call.

Next the engineer creates the `ferry-miles` variable using `create-variable`. Again, this is not just a syntactic change; the operation verifies that it does not mask or conflict with any existing variable declarations. This new variable is not referenced, so it does not affect the system's meaning. Up to this point, the program performs exactly the same function as before.

Finally, the engineer augments `total-miles` by adding code that sums `rolled-miles` with `ferry-miles` and initializing `ferry-miles` to zero. The remaining changes are enhancements that involve introducing the `ferry` module (See Figure 1.4).

The tool eased the structural change of splitting the two miles concepts by creating the `total-miles` function and finding all uses of the original expression, but let the engineer decide which uses represented the abstraction `total-miles`. The actual enhancement—adding ferry miles to rolled miles throughout the system—was simplified by encapsulating the total-miles concept in a function. This localized the travel concept: only one addition was required to incorporate ferry miles. Another positive byproduct is that the `total-miles` function structurally represents that it is built from smaller components, preventing direct modification of its value.

Without a restructuring tool, the repair requiring the fewest changes is augmenting each reference of the original `total-miles` variable to include a reference to ferry miles. This would minimize the number of changes that the engineer manually performs, in the short term minimizing the cost of maintenance and the chances of introducing an error, but also introducing coupling by distributing several identical expressions throughout the program. Another change to the miles concept would require changing all those expressions again, unlike the restructured version, which has only a single instance of the expression.

Although restructuring by hand can improve structure to a degree, the process is error-prone, as observed in Chapter 3, degrading the program by adding to the change strata. Since the restructuring tool automates meaning-preserving global changes, it makes the necessary non-local changes without error.

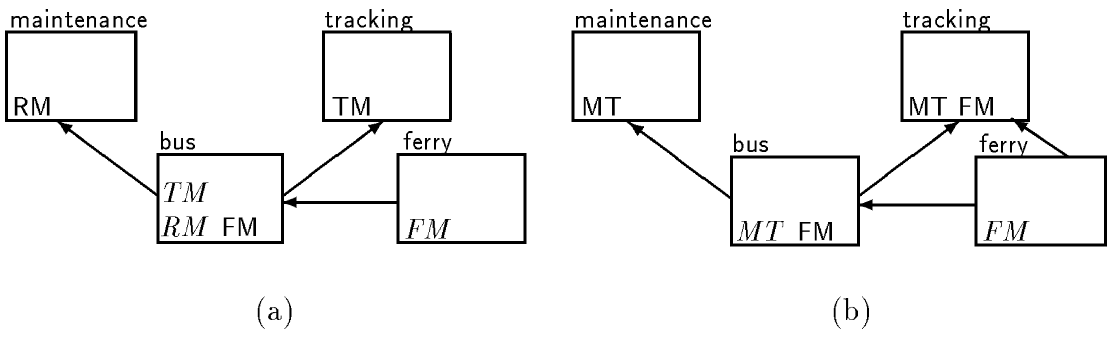


Figure 1.4: The module export–import structure of the extended system with restructuring (left) and without (right) (RM = `rolled-miles`, FM = `ferry-miles`, TM = `total-miles`, MT = `miles-traveled`, definitions in italics)

1.6 Structure and Meaning

The primary challenge for automating meaning-preserving restructuring is formulating a model for correctly and efficiently implementing meaning-preserving structural changes. A standard way of describing a meaning-preserving structural change is through an algebraic law, such as the distributivity law in algebra:

$$x \times (y + z) \equiv (x \times y) + (x \times z)$$

The property allowing this algebraic description is *transparency.* Multiple references to the same symbol (say, to x above on the right-hand side) denote the same value—symbol and value are immutably associated. A program written in a functional language has transparency because it has no variables for representing changing state. This allows algebraic manipulation of functional programs.

C.A.R. Hoare *et al.* demonstrated that imperative programming languages also obey powerful and intuitive algebraic laws that permit source-to-source transformation [Hoare et al. 87]. For example, there is a law that says a variable reference can be replaced by its defining expression. For example, given expression `E`, and expression `F` using `x`, `F(x)`, then `x := E; F(x)` is equivalent to `F(E)`.

A law as simple as this applies only to languages with restrictions on input/output, pointers, recursion, and procedure call to achieve a degree of transparency that normally is not present in imperative languages. In the example above, `E` and `F`, by definition of the language used by Hoare et al., are known not to have side-effects. This is important because the order of evaluation is potentially changed by substituting `E` in place of `x`. Also if `x` is referenced multiple times in `F(x)`, then `E` is evaluated multiple times in `F(E)`. If `E` were allowed to contain side-effects, the substitution would cause repeated side-effects, disallowing application of the law. Further, in an imperative programming language with global variables and procedure call, meaning-preserving substitution depends on how global variables are referenced in the procedure calls made in the expression. In this case the properties of an expression cannot be locally determined.

These problems of imperative programming languages can be overcome by a richer notation that captures program properties not immediately visible in the text. The solution adopted in this thesis is a notation based on control and data flow relationships, represented by the program dependence graph (PDG) [Kuck et al. 81][Ferrante et al. 87]. Among other important properties, the PDG has more of the referential transparency necessary to reason algebraically about a program. This is achieved by representing a variable by edges that explicitly carry values between operations, which are the vertices of the graph. As described in Chapter 4, when combined with a conventional hierarchical syntactic abstraction of the program, it is possible to reason algebraically about structural changes to an imperative program, and to implement those changes efficiently as well.

1.7 Goal

This chapter has introduced the basic motivation and ideas for automated restructuring, including examples of how it can be used, a cost model for maintenance and some background on manipulating structure. Together these imply that automated restructuring can be beneficial if successfully implemented. The ultimate goal of this research is to show that meaning-preserving transformation can restructure, substantially lowering the cost of the changes required to maintain a software system. However, this requires long-term use in a realistic setting, so the purpose of this thesis is to take the first step by demonstrating the following:

Automating the meaning-preserving activities of restructuring through transformation improves the manual process of restructuring. In particular, the automation not only prevents the introduction of errors during restructuring, but allows locally specifying structural changes.

The hypothesis is demonstrated by (1) defining a set of transformations that can restructure programs, (2) conducting an experiment with a restructuring tool and human subjects to demonstrate the problems with manual restructuring, (3) defining a model for meaning-preserving source-to-source transformation, and (4) using the model to efficiently implement the transformations as a tool. The following is a more detailed summary of the contents of the thesis.

1.8 Overview

Chapter 2 describes the basic meaning-preserving transformations necessary for restructuring programs written in a block-structured imperative programming language, in this case Scheme [Dybvig 87]. To show how the transformations restructure, they are used in the prototype to restructure a Scheme matrix multiply program.

Scheme was selected because of its rich imperative features, its simple syntax, and the availability of a PDG package for Scheme programs [Larus 89]. The implementation is in

Common Lisp because of its prototyping flexibility and because it is the implementation language of the PDG package. There is no inherent or conceptual limitation in applying the ideas of the tool to other more commonly used languages, and the transformation set extends naturally to the manipulation of module and class interfaces, which is discussed in Chapter 7.7.

In Chapter 3, matrix multiply is again used in an experiment to evaluate the benefits of automated restructuring. The experiment records the performance of several programmers, each restructuring the matrix multiply program by hand. The observed manual process is compared with the process of using the tool. The comparison reveals that manual restructuring is haphazard and error-prone. It also indicates that the style of restructuring supported by the tool is consistent with manual restructuring techniques, implying that the tool should be intuitive to use.

Chapter 4 defines a model for building efficient, meaning-preserving transformations. It is critical to have assurance that the transformations in the PDG (and in the program text) preserve meaning. It is also important that a meaning-preserving transformation be straightforward to design and implement. One problem is that the program represents scope structure well, but the PDG represents the semantic relationships between expressions well. The solution to this dichotomy is to create an invertible mapping between the program and its PDG so that the best features of each are equally accessible. This relationship is formally defined with a commutative diagram. Reasoning about meaning-preserving transformation on the program is potentially difficult because the changes are physically distributed in the program. However, the equivalent changes in the PDG representation are local. The commutative diagram facilitates justifying a transformation in the PDG, and mapping it to an equivalent source-to-source implementation by a process called *globalization*. Once both a program and equivalent PDG transformation exist, the commutative diagram justifies applying them together, avoiding the expense of reconstructing one representation from the other.

The following chapter on implementation describes how the model guided the design of the prototype restructuring tool. This chapter demonstrates how the mathematical properties of scopes, the PDG and the commutative diagram are exploited to simplify implementation and achieve good performance. It also describes the use of program integration techniques that allowed prototyping research ideas in this thesis.

Chapter 6 evaluates the research by comparing this work with other research in the areas of transformation, restructuring and maintenance. Although there is a significant amount of work in program dependence graphs and also in transformation, the two have not been applied together in a maintenance context. Chapter 7 discusses the current limits of meaning-preserving restructuring as a technique, and as realized in this thesis. This thesis has explored restructuring in a language without modules, and has not restructured large programs, but solutions are proposed for these shortcomings. The primary weakness of the prototype is poor performance because the PDG is entirely reconstructed from the program after most transformations. The incremental update techniques described in Chapter 4.5 can overcome this anomaly of the prototype. To close the thesis, the contributions of the research are summarized and evaluated.

The appendix presents a larger and more involved program restructuring using the prototype tool. The program, the KWIC indexing system, is a well-known example introduced by Parnas for discussing issues in module decomposition [Parnas 72]. The restructuring derives Parnas's preferred decomposition from one he showed is inferior.

Chapter 2

Transformation–Based Restructuring

Program transformation is the approach taken in this thesis to restructuring a program. A transformation manipulates a program to create a new program. Meaning-preserving transformations are an important class of transformations, because they can change the appearance or speed of a program without affecting its input/output behavior. To show how meaning-preserving transformations can restructure a program to improve maintainability, this chapter introduces a taxonomy of structure, analyzes several transformations with it, and then applies the transformations in two examples to show how they are used to restructure.

2.1 The Global Transformation Paradigm

The transformations in the tool have several basic properties. First, when a transformation is applied by the engineer, it is guaranteed either to succeed and produce a new program with the same meaning as the initial program, or else to fail and leave the program unchanged. Second, the engineer applies a transformation to a syntactic construct. Third, to preserve the meaning of the program, the tool may make non-local changes to compensate for the syntactic change. Together these properties assure preserving the program's meaning and free the engineer from the work involved in the updates. In essence, the tool is a semantics-preserving structure editor, allowing the engineer to focus on the design aspects of the restructuring.

To meet these goals the transformations for restructuring described in this thesis follow a strict paradigm. First, a transformation does not modify any operation in the program whose output might be different from its input (basically no operator other than assignment), nor the data or control flowing into them. This assures that the data flowing out of each operation when it is executed remains unchanged, and so the meaning of the program is unchanged. What a transformation *can* change is how values are delivered from one operation to the next. In particular, only constructs involving assignment, sequencing, or the visibility of variables can be transformed, such as scoping and procedure constructs. These organize computations but do not actually perform them. For example, a variable reference can be replaced by the expression that gives a variable its value. On the other hand, the paradigm does not support compensating an increment of a variable reference by a decrement of its defining expression, since + and - are value-changing operations. A more rigorous model for this paradigm is presented in Chapter 4, and possible shortcomings and extensions in Chapter 7.

Another aspect of this paradigm is how global restructuring is achieved, although it is locally specified by the engineer. According to the restrictions above, value-changing operations cannot be modified. So, for example, the only changes to a procedure that will be allowed involve the manipulation of its name, its parameters, and the location of the computations within its body. None of these changes alone is likely to preserve meaning, so each implies particular updates to the uses of the procedure. Changing the procedure's name implies changing the names of its uses to correspond; changing the order of parameters implies changing the order of arguments in the requisite calls, and moving an expression within the procedure to outside it implies adding a parameter to the procedure (and an argument to each call) to pass the value of that expression in when called. Of course, checks may be required to assure that these changes to the uses are sufficient to preserve meaning. The globalization paradigm, then, exploits the link between a definition of an abstraction and its uses. Since there are normally many uses for a single definition, it is natural that for most transformations that the engineer's change is specified on the definition, and the uses are updated by the tool. However, in some cases the roles may be reversed if the transformation is to disassociate the relationship of a particular use from its definition, say by inlining the abstraction at the point of the use.

Is there a finite set of such minimal transformation functions for a programming language? A programming language has a finite set of syntactic forms—forms that relate computations by managing variable name spaces and delivering values. For example, Scheme, the language supported by the prototype described in chapter 5, has expressions for generating values (procedure call and variable reference), variables for holding values, procedures for encapsulating computation (`lambda`), an expression sequencing construct (`begin`), scope bindings combined with expression sequence constructs (e.g., parameter passing, `let`, `let*`, `letrec`), and assignment (e.g., `set!`, `set-car!`, `set-cdr!`). To each type of syntactic form there are only a few things that can be done to it that can be compensated without changing the operators in the program. Loosely, these can be finitely enumerated as the set of syntactic form types F that have a semantic relationship with another syntactic form type s such that there exists transformation $t : s \rightarrow f, f \in F$ without changing the operators in the program. For example, both a variable and an expression can produce a value. If the variable is assigned the value of the expression, then the variable returns the same value as the expression. Perhaps one can be used in the place of the other. By counting transformations in this way, for the N syntactic forms in a programming language there are at most N^2 transformations. This includes transformations that transform a syntactic form to itself, perhaps modifying the order of objects within it.

Although it appears that there might be an arbitrary number of transformations of an object on to itself, these collapse into one transformation. For example, for a parameter list of length n, there are $(n(n-1)/2)-1$ possible reorderings of the parameters. However, these are all reachable by the `move-param` transformation, a generalization of the `swap-parameter` transformation used in Chapter 1.3. Another way that there may appear to be an infinite number of transformations is by not choosing *minimal* transformations. For example, suppose a transformation that takes the variables in a call and binds them to let bindings, and then makes the call with the new bindings:

```
(f x) ⟶ (let ((x1 x)) (f x1))
```

and one that creates *two* levels of indirection:

```
(f x) ⟶ (let ((x1 x)) (let ((x2 x1)) (f x2)))
```

and so forth. The latter is not a different transformation, however, just two applications of the first transformation.

But is the resulting set of transformations is sufficiently powerful to allow localizing (most) any design decision so that it may be changed at lower cost? Intuitively, following the reasoning above, the set would be complete if it covered the N^2 possibilities. More quantitatively, to localize any property means being able to colocate any subset of program components within a module. A proof of whether a set of transformations can do this seems untenable, since it requires precise knowledge of the transformations, the programming language, and even the program. However, the transformations to be shown below have been derived in the process of restructuring programs by hand and with the tool. Some of these have also been suggested as useful in the literature [Burstall & Darlington 77][Hoare et al. 87]. However, the effectiveness of this set in localizing change is being validated in practice. Also, although the current set is not complete, it is not difficult to add new transformations as experience with restructuring grows (see Chapter 5.5).

2.2 Transformations in Compiler Optimization

Many of the transformations for restructuring introduced below are well-known compiler optimizations or their inverses.[1] For example, inlining a function is a common optimization, and its inverse, function extraction, was applied in the transit example and also in the restructuring of matrix-multiply shown later in this chapter. The need for flow analysis (see Chapter 4) and preserving meaning are common threads, too. But there are several differences between compiler optimization and program restructuring for aiding maintenance. Most obviously, the motivations are different. As discussed in the previous chapter, the evaluation function for choosing the best restructuring for lowering maintenance costs is subjective, being dependent on unknown future changes. Also, it is not clear that there is an inexpensive algorithm for choosing the transformations that will achieve a particular structure, although in practice it does not seem to be a problem when the transformations are chosen interactively by the engineer. The cost function for optimization is clear—lower program execution time—although it is still hard to evaluate off-line since performance is input dependent. Thus user interaction is not as necessary in optimization as it is in restructuring. Also, optimization is typically not required to produce a valid or readable source language representation, working instead on intermediate code or assembler, a simpler task. Exceptions are the transformations described by Loveman [Loveman 77], and recent results in optimizing register accesses in loops [Callahan et al. 90]. Still, neither of these require a readable source, nor an interactive user interface.

Another difference is that most optimizations are, in practice, local in character—manipulations on loops and basic-blocks.[2] An exception is the manipulation of procedure definitions to optimize parameter passing from caller to callee. Even in this case the callee definition must be copied so that changes to it for the caller do not affect other callers of the same routine.

[1]It has been argued that many compiler optimizations are based on well-known manual programming techniques for optimization. Perhaps the influence has come full-circle.

[2]D. Callahan, Personal Communication, 1991.

Other domains using program transformation are discussed in Chapter 6.

2.3 Techniques for Representing Structure

By understanding the ways structure can be expressed with modules, the usefulness of a transformation can be shown by demonstrating how it changes modules to modify structure. Harold Ossher [Ossher 87] described the basic structural properties essential to organizing information in a concise fashion, allowing substantial reuse and localization of changes.

Grouping identifies a set of program components as being part of an aggregate component, perhaps with a name. Denoting a group is to denote all the components within it. Typical grouping constructs in programming language are the expression sequencing construct, function body, and module body. *Abstraction* allows identifying a (possibly grouped) component through a protocol that hides the details of the internals. Procedures and modules fall into this category. A procedure's protocol is a name and a list of parameters to be processed by the named function. A module's protocol is the union of its procedures' protocols (with implicit or explicit constraints on how they may be ordered). Abstraction is useful because it allows easy reuse of a potentially complex program component through a simple interface, and there is only one use of the component's internals—the definition. This means that a single change to the definition automatically propagates to all uses without any further change—as long as the interface does not have to be changed as well. *Analogy and Deviation* constructs a component by exploiting its similarity to an existing component, and then adding some things to account for differences. An example is wrapping a procedure with another to modify its output. Using inheritance to help build a subclass is another. Closely related to deviation is *approximation*, which is defining a component that similar to what is desired, but not quite. The desired component can be created through deviation, and the approximation can be reused to help create other components. A superclass is an example of an approximate component that can be reused through the deviation-process of subclassing.

These organizational techniques can be harmful as well as helpful, so being able to remove these structures as well as add them is essential. For example, if two components are grouped together, but it is desired to denote just one, grouping defeats the goal. Also, an abstraction may not have a general enough protocol to allow controlling essential parameters. Any of these problems could result in reimplementing most of the same component, introducing redundancy and thus decreasing the locality of change if the common parts must ever be enhanced or repaired. Alternatively, deviation might be applied to solve these problems, which in these instances would obfuscate the true relationship between the overly specific component and the resulting more general one.

Deviation, like grouping and abstraction, can be overconstraining, in this instance because it establishes a tight binding between the base and derived objects. A change in the base is likely to affect the derived object. If this is not the desired effect, then both the base and derived objects will need to be modified, which is non-local change.

Thus there are two basic structural problems, both of which force the change of a single concept into updates on multiple program components, meaning that changes are not local. The first problem is undesirable coupling (See Chapter 1.4.1), say perhaps due to redundant codings of a computation in different modules. If one component has to be changed, it is likely the other must be changed as well: modifying a single 'concept' requires multiple updates across modules. Abstracting the relationship, if it is due to redundant coding, into a single function definition converts the coupling into a cohesive definition. In other circumstances, undesirable coupling may be converted into the more desirable data coupling—accessing data through parameter passing.

The other problem is that two components that are not strongly related are in the same module anyway—they are only weakly cohesive. This means that changing one potentially impacts the other, and the effect may not be desired. To avoid propagating the change to the other requires performing not only the initial change, but some compensating change to undo the effect on the dependent object: again, updating a single concept requires multiple updates. This false relationship can be eliminated by removing the offending grouping, abstraction, or deviation of the components.

To address these two problems, then, the transformations must modify structure by creating, modifying, or removing syntactic constructs that implement these structuring techniques. For example, in the transit example the `extract-function` transformation created a procedure abstraction of the concept `total-miles`, which was implemented by returning the `miles-rolled` variable. This localized the addition of ferry miles to `miles-rolled`, since the only reference to `miles-rolled` was within `total-miles`.

2.4 Transformations

This section introduces the most useful of the transformations necessary to restructure programs, and evaluates them with regard to the structuring criteria of the previous section. Rather than give isolated examples of their use, most are used in two examples presented after this section. The first shows how data representation transformations can generalize a data structure, and the second uses control–oriented transformations to relayer a Scheme matrix multiply program. These show how transformations are used together to restructure. The matrix multiply example also will be used in Chapter 3 for examining the qualitative differences between tool-aided and manual restructuring.

Moving an expression. Moving a program component is perhaps the most common transformation—usually as part of another transformation, but also on its own via `move-expr`. When moving an expression, there are no compensating transformations, just checks. In particular, the bindings of the variables referenced in the expression cannot change; the movement will change the order of evaluation of some expressions, which must not change the values returned from expressions; finally, the moved expression must be evaluated in the same circumstances as before.

Moving an object closer to others, specifically when it is being moved between scopes, is regrouping. Also moving an object next to another will allow them to be grouped by another transformation.

Renaming a variable. Transformation `rename-variable` takes a variable binding and a new name, and renames the variable (and all its uses). The transformation must check at each use that the name does not conflict with any existing names.

Renaming is important to structure, for example, because it may be desirable to group two objects of the same name that previously were in separate scopes. One must be renamed to make this possible. It was used in the transit example to give a more precise name to the variable `miles-traveled` to reduce ambiguity with respect to other names being used, such as `ferry-miles`.

Inlining an expression. The transformation `var-to-expr` basically inlines an expression, replacing the uses of a variable definition with the defining expression. The engineer selects the assignment to be inlined and the tool handles finding and inlining the uses. An alternative version of the transformation takes a single variable use and inlines only the single use, not others from the same definition. Since the expression is moving, it must satisfy all the conditions for a move to each of the variable uses. Additionally, if the result will allow multiple evaluations of the expression, the transformation must assure that there are not side-effects in the expression (since repeating the side effects could change the meaning of the program). Also, if there are multiple uses to be inlined, the expression cannot, in general, have side-effects. Finally, if a use has two potential definitions—this can occur with a conditional assignment—the transformation is prohibited.

A more powerful version of `var-to-expr` is `binding-to-expr`. Given a variable binding by the engineer, the transformation finds all the assignments to the variable and performs `var-to-expr` on each. When this is applied to a function definition's parameter, the tool must check that all the calls of the function use the same expression for the argument being inlined. This is because each call represents an independent binding of the parameter, but when inlined all the binding expressions must be merged into the one inlined expression.

Both versions allow specifying whether the variable binding should be deleted (if possible). The flexibility is important because it is not always clear that it should be deleted. The engineer may have another use for the variable after the transformation is complete.

Inlining removes abstraction. It generates multiple instances of the abstracted object, allowing individual instances to be modified without affecting others. More specifically, the version of `var-to-expr` that inlines a single use disassociates that variable use from the others that get their value from the same expression evaluation. After applying this, it is possible to change the inlined instance without affecting the other uses. The version that inlines function bindings not only inlines the parameter, removing its abstraction, but it also narrows the interface of the function abstraction.

Abstracting an expression. Given an expression by the engineer and a name and location for a binding, `expr-to-binding` performs roughly the inverse of `binding-to-expr`, moving an expression into a scope binding, assigning the result of the expression to the new binding variable, and putting a reference to the variable in the old location of the expression. As above, this can be successful only if the expression at the new location would have the same value as in its original location. It also requires that the newly defined variable binding not conflict with the scope of any existing bindings of the same name.

This transformation has a broader impact when taking an expression from inside a function definition and abstracting it as a parameter of the function. This means that the removed expression must be passed as an argument to all calls on the function. The value of this is that the resulting function is more general—different values other than the original embedded one can be passed in newly added calls.

Abstracting an expression into a procedure binding has the added benefit of allowing different expressions to be bound to the new variable, even though normal expression abstraction disallows multiple expressions to be bound. This is because a procedure disassociates its declaration of the computation from its execution, so its parameters benefit similarly because their declarations are separated from their bindings.

Extracting a function. The transformation `extract-function` turns a sequence of expressions into a function, and replaces the extracted statements with a call on the function. Again, this is a more general version of expression abstraction, however, since a function can take parameters, any variables that would not be defined in the new location can have their values passed in to the procedure as arguments. The engineer provides the expressions, a name for the function, the expressions or variables to be made parameters, the names of the parameters and the location of the new function. `extract-function` is implemented by first creating the function inline, and then doing an expr-to-binding. For this transformation to succeed requires that the right parameters get abstracted so that moving the function to a new location does not leave any free variables.

This has the same benefits as expression abstraction, but with more potential for reuse, and hence localization. As already mentioned, `extract-function` was used in the transit example to localize the reference to `miles-rolled` inside the function `total-miles` so that modifying `miles-rolled` to include ferry miles localized with `miles-rolled`.

Procedure abstraction is reversible with `inline-function`, so it can remove undesirable cohesion.

Scope-wide function replacement. This transformation `scope-sub-call` replaces repeated sequences of the code of an existing function with calls on the function. This is often used following function extraction.

To find the repeated codings requires a program equivalence test, which in general is infeasible [Downey & Sethi 78]. However, there are conservative techniques that are fast but can abstract away many anomalies [Yang et al. 89] (See Chapter 4.3). As a back-up, an optimistic heuristic technique can be used, such as comparing the usage of a candidate expression with the usage of the function that is being substituted. That is, if each is used as the k^{th} argument of a call on the same function, or at least if the uses of each require the same type, then there is some narrowing of the search. Of course, an optimistic heuristic result requires approval of the tool's choice by the engineer.

Matching is also complicated by the fact that when trying to match a function to an expression, the function's parameters must be matched against actual code. That is, to match a function to inline code, an inferencer must try to select which parts match the function body, and which parts should be passed as parameters. The matching requires some kind of logical inference, although this is not straightforward since two references to a function parameter can have different values, due to side-effects. This means that function parameters cannot be treated strictly as logical variables.

Scope-wide replacement requires user approval of each substitution also because two semantically identical expressions do not necessarily mean that they represent the same abstraction. For example, in the transit example, instances of `miles-rolled` were to be replaced with calls to the semantically identical `miles-traveled` function. However, the references to `miles-rolled` in the maintenance module were to remain unchanged, since they did not (conceptually) represent total miles traveled, but only those miles that resulted in wear-and-tear on mechanical components. This required allowing the engineer to filter matches. Comparing the usage of matching expressions, as suggested in the previous paragraph, works well here since semantically identical expressions that are the same abstraction will tend to be used in a similar way.[3]

Scope-wide function replacement has the same localization benefits as function extraction, but it can help recover structure *after* redundant coding has been introduced already. `extract-function` can be used only to avoid redundant coding.

This transformation is a slight departure from the paradigm of local change by the engineer compensated by the tool. There is no local change, and so there is no necessary compensation—thus the need to prompt the engineer for each substitution. What is really changing is the engineer's perceived structure of the system and how it is best rep-

[3]Alan Demers (Personal Communication, 1991) suggested this technique for eliminating spurious equivalent matches. The problem posed was removing the embedded uses of the file number `1` in C programs. Since C programs use `1` for incrementing integers, incrementing structure pointers, and as a file number for the standard output, there are too many matches for the engineer to interactively filter. However, restricting the tool to selecting matches that only use the number `1` in `write` is sufficient to differentiate those uses that should be abstracted as file numbers.

resented. The tool helps by finding all the candidates and assuring that the substitutions preserve meaning.

Adding reference indirection. There are also data representation manipulations, such as aggregating a group of variables into a record, or changing a single variable into a list of variables, and making the requisite updates to the uses of the modified structure. There are two such types in Scheme, vectors (arrays) and lists (linked lists). The corresponding transformations are `vectorfy-bindings` and `listify-bindings`, which take a name and a list of variable bindings provided by the engineer and create the requisite object and replace the references to the old bindings with references to the new structure.

The benefit of creating a record is its grouping effect—it explicitly joins the allocation of previously independent elements, and gives them names that explicitly relate them.

Converting a scalar into a list—and later putting references to it in a loop—has the benefit of allowing multiple instances of an object where before only one was allowed. This is grouping, and abstraction as well, since any instance in the list can be used in place of another.

Adding looping to list references. If a scalar variable has been converted into a list with `listify-bindings`, it is often natural to introduce iteration over the list with `loopify` to ease enhancing the list to contain multiple elements. Since the lists are single-element, there is just one "iteration", so meaning is not changed. `loopify` works for vectors as well.

The benefit of this transformation is that it enhances the control structure to access all the members of the group—list—in sequence, rather than just a single element.

2.5 A Data–Oriented Restructuring

This section highlights these last two data-oriented transformations. The tool commands shown below are slight variations of the actual commands used in the tool; the variations generally make the commands more readable by giving understandable names (shown in <angle braces>) to program locations. The command syntax is Common Lisp, but they could be expressed naturally as window- and mouse-based commands with a more developed user interface. One peculiarity of Common Lisp is the use of *keywords* to denote special parameters to a function. For instance, invoking

`(move-expr` e `:before` g`)`

means calling transformation `move-expr` with its first argument as e and its `before` argument as g.

Below is a piece of a program that creates a picture element, puts it into a scene, and then flushes the object to the screen. It is desired to enhance this code so that multiple elements are displayed by this code.[4]

```
(let ((object (picture-element 'square)))
  (put object world)
  (display object world))
```

To allow multiple elements, first the scalar picture element is transformed into a list of picture elements. The engineer executes the command `(listify-bindings <object binding> 'objects)`. This transforms the `let` binding of `object` to be a list containing the original value bound to `objects`, and the uses are changed to dereference into the first element of the new list:

```
(let ((objects (list (picture-element 'square))))
  (put (car objects) world)
  (display (car objects) world))
```

[4]The Xerox Mesa MultipleInstance tool, for example, when given a tool rewritten in a particular style, supports multiple instantiations of the tool, including process context switching, locking, and window management. The process of modifying a tool to be acceptable to MultipleInstance requires, among many other tasks, redeclaring the global variables so they are captured in a single record type so that they can be instantiated multiple times in a list. The restructuring being performed here is essentially this preparation, plus the addition of the looping.

For `listify-bindings` to succeed, the name of the new variable `objects` must not conflict with any existing variable name in the same scope. Although this particular application of `listify-bindings` is not particularly global, in general it is applied in the entire scope of the changed variable, which could comprise entire text of the program.

Now that there is a list structure for handling multiple picture elements, the engineer can now transform the body of the `let` to be looped over so that all elements of the list are automatically accessed. The tool user executes the command:

```
(loopify <objects> <body let> 'objs)
```

This transforms the body of the let to be a `do` loop over the body, with `objects` being iterated over and `objs` being the name of the iteration variable. The result is:

```
(let ((objects (list (picture-element 'square))))
  (do ((objs objects (cdr objs)))    ; looping clause
      ((null? objs) (car objects))   ; termination clause
    (put (car objs) world)           ; loop body
    (display (car objs) world)))
```

Now the code, which performs the exact same task as before, is much easier to modify for adding multi-element function. For `loopify` to succeed, the new variable name `objs` must not conflict with any existing variable in the same scope. Also, the list being looped over must have only one value in it, because otherwise there would be extra iteration that was not present before.[5]

2.6 Restructuring a Matrix Multiply Program

Now to show how the remaining transformations are used, and give a better sense of how restructurings are achieved, a matrix multiply program [Dybvig 87], shown in Figure 2.1, is restructured. Matrices in this program are represented as lists of (equal length) vectors,

[5]The conditions stated for transformations are conservative, and often can be generalized to improve the likelihood of successful transformation, without affecting correctness. The over-conservativeness made implementation of the prototype simpler. For example, `loopify` could be made to work on multi-element lists if the extra iterations cause no side-effects. In some instances here, the conditions stated are simplifications to make the *explanation* easier.

although the representation is hidden from the multiplication function through the use of auxiliary functions.

The restructuring centers around three local functions and an implicit, inlined function embedded in the main function. Extracting a form of these functions may prepare for later functional changes or for reuse by other programs. The first two local functions, `matrix-rows` and `matrix-columns`, respectively report the number of rows and of columns in a matrix. The third local function, `match-error`, reports an error if the two matrices do not match in size. This function also has an embedded constant (`'matrix-multiply`) in it for reporting the name of the function that received the incorrectly sized matrices. All of these functions, if at the top-level, could be reused to implement, for example, a `matrix-add` function. Finally, the part of the inner loop of the matrix multiply that computes the inner-product of a row of one matrix and a column of another is an operation that could be extracted and invoked as a separate function.

The restructuring tasks are:

- Move the three local functions to the top-level.
- Remove the `letrec` that contained the local functions.
- Modify `match-error` to accept a parameter that is the symbol-name of the function that received the mismatched matrices.
- Make the inlined inner-product function a callable, top-level function that computes the dot product of a row of one matrix and a column of another. The parameters to the function are to be the two matrices and the respective row and column. The code in the inner loop that computes this in the original program is to be replaced by a call to this new function.

The computer-aided restructuring of the matrix multiplication program is relatively straightforward. The first step in the restructuring is to add a parameter to generalize `match-error`. The command

```
(expr-to-binding <'matrix-multiply> 'header :scope <match-error>)
```

```
(define 1+ (lambda (x) (+ x 1)))

(define make-matrix (lambda (rows columns)
  (do ((m (make-vector rows))
       (i 0 (1+ i)))
      ((= i rows) m)
      (vector-set! m i (make-vector columns)))))

(define matrix? (lambda (x)
  (and (vector? x) (> (vector-length x) 0) (vector? (vector-ref x 0)))))

(define matrix-ref (lambda (m i j) (vector-ref (vector-ref m i) j)))

(define matrix-set! (lambda (m i j x) (vector-set! (vector-ref m i) j x)))

(define matrix-multiply (lambda (m1 m2)
  (letrec
    ((match-error (lambda (what1 what2)
       (error 'matrix-multiply "~s and ~s are incompatible operands"
              what1 what2)))

     (matrix-rows (lambda (x) (vector-length x)))

     (matrix-columns (lambda (x) (vector-length (vector-ref x 0)))))

    (let* ((nr1 (matrix-rows m1))
           (nr2 (matrix-rows m2))
           (nc2 (matrix-columns m2))
           (r  (make-matrix nr1 nc2)))
      (if (not (= (matrix-columns m1) nr2))
          (match-error m1 m2))
      (do ((i 0 (1+ i)))
          ((= i nr1) nil)
        (do ((j 0 (1+ j)))
            ((= j nc2) nil)
          (do ((k 0 (1+ k))
               (a 0 (+ a (* (matrix-ref m1 i k) (matrix-ref m2 k j)))))
              ((= k nr2) (matrix-set! r i j a))
            nil)))
      r))))
```

Figure 2.1: The Initial Matrix Multiply Program

extracts the `'matrix-multiply` constant from the original body and makes it a parameter, named `header`, of function `match-error`, and updates each call on `match-error` to pass `'matrix-multiply` as a parameter. The command checks that the abstracted value has the same value in its new context as it did in the old and that `header` is an acceptable name in the scope created by `match-error` (that is, there is no existing parameter or local variable of the error function with that name). In this case, the command succeeds and transforms the `match-error` function to

```
(match-error (lambda (what1 what2 header)
  (error header "~s and ~s are incompatible operands" what1 what2)))
```

and updates the (only) call to

```
      (match-error m1 m2 'matrix-multiply)
```

The second step is to move the three local functions to the top-level and to remove the `letrec` that encloses them:

```
(move-expr <match-error> :before <matrix-multiply>)
(move-expr <matrix-rows> :before <matrix-multiply>)
(move-expr <matrix-columns> :before <matrix-multiply>)
(ungroup <letrec>)
```

The first three commands check to make sure that there are no name conflicts in the new scope; in this case they succeed and move the three functions to the top-level, in front of the definition of `matrix-multiply`. The last transformation removes the now empty and useless `letrec` from the body of the matrix multiply.

The last part of the restructuring, extracting the inlined inner product, is the hardest. There are three impediments to performing the restructuring directly.

First, the variable `nr2` is used in the inner product computation but is defined by the enclosing main function; to allow the extraction of the inner product, the value of `nr2` must be made available in the newly extracted function. Second, the value that the inner loop computes is stored in the variable `r`; it must be returned as the value of the extracted function. Third, the inner product is not parameterized; before extraction, it must be explicitly parameterized by the two matrices (`m1` and `m2`) and by the row and column indices (`i` and `j`).

To handle the first problem, `nr2` must be split into another equivalent expression (called `len`):

```
(var-to-expr <second nr2 reference>)
(expr-to-binding <result of previous> 'len :scope <do>)
```

As before, the tool first checks that the new names will not conflict with others in the designated scope and also that the recomputation of the binding produces the same value as before, and causes no extra side-effects. The checks succeed, and the command transforms the inlined inner loop of `matrix-multiply` to:

```
(do ((k 0 (1+ k))
     (a 0 (+ a (* (matrix-ref m1 i k) (matrix-ref m2 k j))))
     (len (matrix-rows m2) len))
    ((= k len) (matrix-set! r i j a))
  nil)
```

To handle the second problem, the independent part of the result expression must be moved out of the enclosing `do`.

```
(pop-out <do> <a>)
```

This command moves the entire return result of the `<do>`, except for the second parameter (`<a>`), outside of the `<do>`.

```
(matrix-set! r i j
  (do ((k 0 (1+ k))
       (a 0 (+ a (* (matrix-ref m1 i k) (matrix-ref m2 k j))))
       (len (matrix-rows m2) len))
      ((= k len) a)
    nil))
```

Now the inner loop can be extracted.

```
(extract-function <do> 'inner-product
                  :old-new-name-pairs '(<m1> <i> <m2> <j>)
                  :before <matrix-multiply>)
```

The parameter `:old-new-name-pairs` is the list of variables or expressions to be abstracted as parameters to the new function. New names can be supplied as well, but here are defaulted to their current names. Note that if the engineer had tried to apply this transformation without moving out `nr2` and the value of the computed inner product, the tool would have aborted the `extract-function` transformation, returning the error message:

```
Variables nr2, r would be unbound in new context.
```

This completes the restructuring of the program, which is shown in Figure 2.2 with the unchanged functions omitted.

2.7 Summary

The key properties of transformations are that they preserve meaning and that globalization of the engineer's local change is achieved by the relationship between definitions and uses of the object. For completeness, the transformations should span the syntax of the programming language, for each syntactic form addressing the problems of poor coupling or cohesion without changing meaning. Each of the transformations provided addresses one of these two problems, and the affect of each on program structure has been shown. Although not a complete set, the transformations provided have proven sufficient thus far, successfully restructuring the matrix multiply program and other examples.

Note that the design of the transformations themselves apply the structuring techniques from the taxonomy to derive one structure from another. For example, abstracting a new parameter for `match-error` created a deviation from the original `match-error`, but the original was destroyed in the process, requiring updates to the uses to preserve meaning.

Some transformations, although they do not appear to be restructuring qualitatively, when put together achieve the goal of restructuring. The restructuring can be viewed as an abstract diagram (See Figure 2.3), showing that hierarchical and grouping relationships among components is substantially changed. A box, as an agent, can only see the

```
(define match-error (lambda (what1 what2 header)
  (error header "~s and ~s are incompatible operands" what1 what2)))

(define matrix-rows (lambda (x) (vector-length x)))

(define matrix-columns (lambda (x) (vector-length (vector-ref x 0))))

(define inner-product (lambda (m1 i m2 j)
  (do ((k 0 (1+ k))
       (a 0 (+ a (* (matrix-ref m1 i k) (matrix-ref m2 k j))))
       (len (matrix-rows m2) len))
      ((= k len) a)
    nil)))

(define matrix-multiply (lambda (m1 m2)
  (let* ((nr1 (matrix-rows m1))
         (nr2 (matrix-rows m2))
         (nc2 (matrix-columns m2))
         (r (make-matrix nr1 nc2)))
    (if (not (= (matrix-columns m1) nr2))
      (match-error m1 m2 'matrix-multiply)
      nil)
    (do ((i 0 (1+ i)))
        ((= i nr1) nil)
      (do ((j 0 (1+ j)))
          ((= j nc2) nil)
        (matrix-set! r i j (inner-product m1 i m2 j))))
     r)))
```

Figure 2.2: The Restructured Matrix Multiply Program

boxes immediately inside it. So on the left, there is only one top-level accessible box, while on the right there are five.

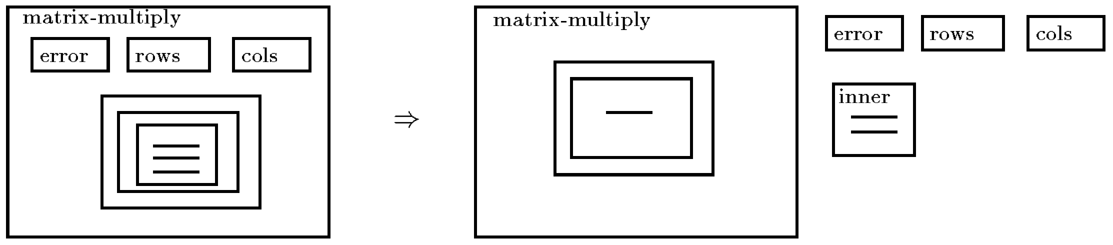


Figure 2.3: Abstract view of matrix multiply's restructuring

What is clear from this diagram is that four objects that were not externally accessible before—all four obstructively grouped inside `matrix-multiply` and one of them not abstracted—are now independent abstractions. The reusability of these four new top-level functions is now much greater because they span a larger scope than one function. As a secondary consequence, the internal structure of the main object is now simpler.

The next chapter presents an experiment that helps evaluate the benefits of tool-aided restructuring, and provides insight on the validity of the tool's mode of interaction with the user.

Chapter 3

Tool-Aided versus Manual Restructuring

The previous chapter introduced several meaning-preserving transformations that effectively separated restructuring from other maintenance activities by making non-local checks and changes to complement the engineer's change. To evaluate this approach to automating restructuring, a small, relatively informal experiment was conducted to compare use of the tool to manual restructuring efforts. Six people were given the matrix multiply program discussed in the previous chapter and asked to perform the same restructuring. An observer watched each restructuring and afterwards interviewed each subject about the process. The results were then compared with the same restructuring as performed using the tool.

One goal of the experiment was to determine whether or not manual restructuring is an error-prone activity. If individuals in fact have little or no problem in restructuring a program by hand, then this is evidence that automating the restructuring process would be of relatively little benefit. Alternatively, if individuals have difficulty with the actual activities (independent of the subjective decisions about what to restructure), this would provide increased motivation for computer-aided restructuring tools.

Another goal of the experiment was to see if the tool's model is consistent with how engineers restructure manually. It is assumed here that the some of the best tools are those that automate an activity that users already intuitively understand.[1] (As an example, consider the success of `make` [Feldman 79]. As another example, consider the lack of success of editors that are strictly syntax-directed.) By monitoring the way individuals restructure a program, it was desired to either solidify or disprove the hypothesis that the tool is natural to the way engineers restructure programs.

3.1 The Experiment

Each subject was presented with the initial matrix multiplication program (Figure 2.1), a description of the four goals of the modification (without using the word "restructuring") as described in Section 2.6 and the motivations for those goals. In addition, the subject was asked to make "some attempt to assure that the modifications are correct; that is, that the modifications have not changed the meaning of the program in any unsatisfactory manner." Provided were a small data file and a Scheme interpreter running on a workstation that handles multiple windows.

After the restructuring, each subject was asked three groups of questions. For the most part these can only be used to detect anomalies in the experiment and collect interesting stories, since even apparently factual questions can have subjective qualities requiring judgments that cannot be quantified. The first questions focused on the modifications themselves. Were they difficult? What were the specific impediments, if any, to the modifications? How did you overcome these? Did you have trouble getting the modification right? Was the cause syntactic or semantic? The second group of questions focused on Scheme and Common Lisp programming ability. Do you regularly program in Scheme? Common Lisp? How would you rate yourself as a programmer in these languages? If you are a Common Lisp programmer, did programming in Scheme present any problems? The final group of questions concerned familiarity with this restructuring

[1]It is also assumed, however, that tools with this property often end up being used in completely unanticipated ways, which is one mark of an extremely successful tool.

work. Have you heard or read about the work? Seen any presentations? Seen the matrix multiply restructuring problem in particular?

The Common Lisp questions were due to the fact that most of the subjects were proficient in Common Lisp rather than Scheme. Although the differences in Scheme and Common Lisp were not significant for this small problem, it was helpful to observe that a programming error was due to confusion about the syntax of Scheme, so as to not give it too much weight in the analysis.

The experiment was introduced to the subject by describing what the program does and how it is done. In addition, two subjects who normally program in Common Lisp rather than Scheme were described the special characteristics of Scheme . (It turned out that only one subject, #5, had ever programmed in Scheme, and that was a number of years ago.) The observer then started the execution of the Scheme interpreter, with the program in a separate window.

During each subject's modifications, the length of the experiment, including the times taken for editing, thinking, running, and debugging were recorded. Also tracked were the order and kind of changes, whether code was copied or reentered, what errors were made (and were they syntactic or semantic errors), etc.

Occasionally during the modifications, the observer would help the subject overcome minor problems with differences between Scheme and Common Lisp. For example, the headers of Scheme functions are slightly different from Common Lisp functions, and some subjects did not notice this immediately by visual pattern matching.

All the subjects were graduate students in computer science. With one minor exception (accounted for by modifying the time taken to do the restructuring), the support tools (editors, etc.) presented no problems to the subjects.

The results of the experiment are summarized in Table 1 and discussion of individual subjects' sessions are given next. (Most of the columns in Table 1 are self-explanatory. The pretest column indicates whether or not the subject tested the original program provided. The prior knowledge column indicates whether the subject had seen the ideas

Subject #	Common Lisp Experience	Time (min.)	Pretest	Errors	Prior Knowledge
1	intermediate/advanced	20	Yes	No	Example
2	intermediate/advanced	20	Yes	Yes	None
3	intermediate	23	No	Yes	None
4	beginner/intermediate	19	No	Yes	Ideas
5	advanced	19	No	Yes	None
6	intermediate	19	No	No	Ideas

Table 3.1: Summary of Data

before, seen the example before, or had no knowledge at all of the work.)

Subject #1. This subject started by testing the initial version of the program. The first change was to paste a copy of the error function at the top-level, modify the copy, move back and update the invocation, and then delete the original. The subject then copied versions of the `matrix-row` and `matrix-column` functions to the top-level, modified them, and then deleted the originals and the associated `letrec` from the main function.

Next the subject typed a new top-level function definition for `inner-product`. The body of the new function was pasted in using the code from the inner loop, and then the names in the body were updated to match the parameters names in the new header. A copy of the `nr2` binding was pasted into the function. Finally, the `matrix-set!` fragment was moved from the main function to the top-level function, and the associated call was placed into the main function. The modified file was loaded and tested successfully.

This subject was the only one who had seen a discussion of the restructuring work that had included the matrix multiplication example.

Subject #2. This subject started by pasting copies of the three local functions at the top-level, immediately modifying them to become stand-alone functions. The subject deleted and reinserted parentheses at the end of each of these functions, using the editor to check that the parentheses were balanced. The subject next ran tests on the original (unmodified) file. Then the subject deleted the original versions of the local functions, and the enclosing `letrec`, from the main function and then tested this version. The subject next added a parameter to the error routine, generalized the body to use this parameter, and then moved to the call site and update the invocation of the error function.

The subject then typed the preamble of the new top-level inner product function and inserted a copy of the inner loop. The `matrix-set!` was deleted from the copy at the new definition, the parameter names for the new function were updated, the call site was modified, and the old loop was deleted.

When testing this version, a run-time error indicated that `nr2` was undefined. The subject first replaced the reference to `nr2` in the new function body with a direct call to the `matrix-rows` function, but then decided to insert a copy of the binding from the main function. The program was loaded and successfully tested.

Subject #3. This subject started by moving the local functions to the top-level, modifying them to be full-fledged functions. The subject then modified the header, then the invocation, and then the body of the error function.

The preamble of the new inner product function was typed. The invocation of the new function was then entered, the loop body cut and then pasted at the top-level. A copy of the `nr2` binding was added to the new function, and the invocation was cleaned up. Then the subject searched for any remaining uses of `nr2` in the main function to determine if the original binding could be deleted.

Upon loading the file, the subject was notified that there was a syntactic error—an extra right parenthesis—at a given line number. After checking a function that turned out not to include this line number, the subject reloaded and the same error occurred. The subject now found the error quickly, tested the new inner product function directly, and then tested the final version of the matrix multiply function. The subject checked the results by hand, since the original version had not been tested.

Subject #4. This subject handled the inner loop modification first, starting by entering the preamble and copying the inner loop (not including the `do`). The subject replaced the use of `nr2` in the new body with a direct call to `matrix-rows`, tweaked the new body, moved to the main function to add the new invocation to inner product and to delete the old loop.

The subject then added the parameter to the header and body of the error function. The three local functions were then copied and pasted at the top-level, after which the subject returned to the main body and deleted the original local functions and their enclosing `letrec`. The subject loaded the modified file, testing the new inner product function and then testing the main function.

When testing the error case for the main function, there was an error because the subject had not added a parameter to the invocation of the error function. This was fixed, and the program was reloaded and retested successfully.

Subject #5. This subject first moved the local functions to the top-level, including the associated `letrec`. The header and then the body of the error function were updated. Then the headers of the two other (now) top-level functions were updated.

The call site for the inner product was updated, and then the inner loop was deleted and pasted at the top-level. The preamble was then typed, the `nr2` binding was moved from the main function to the new function, and the body of the new function was tweaked. A copy of the `nr2` binding in the new inner product function was made and copied back to its original location in the main function, after the subject searched the main function for other uses of `nr2`. The call site of the error function was updated, and the modified version was loaded. There was a basic syntactic error in the header of the new inner product function, which the subject fixed easily. The program was reloaded, tested, and checked by hand.

Subject #6. This subject started by moving the local functions to the top-level, modifying the headers, and deleting the `letrec`. The subject then added the parameter to the header of the error function, next modifying the body and then the associated invocation.

The subject then made an aborted attempt at adding the inner product function, returning to the original version using a backup file in the editor. The subject then entered the preamble and then moved the inner loop (except for the termination return) to the new function. The subject realized that `nr2` was not bound in the new function, added a parameter to pass the value, and changed the reference to `nr2` to match the parameter name. The termination return was then moved to the new function, and the invocation was added in the main function. To determine the actual parameter to associate with the added formal in the inner product function, the subject searched an old version of the file. After including the proper expression for the actual, the subject loaded the file and successfully tested, checking the results by hand.

3.2 Discussion

Overall, the study of the full transcripts of the modification sessions and the subsequent interviews led to several conclusions about how people manually restructure and how this relates to computer-aided restructuring.

Copy/paste and cut/paste. The first conclusion is that the subjects used a mixture of the copy/paste paradigm (where a copy of the original source is made, moved, and modified) and the cut/paste paradigm (where the original source is moved in its entirety). In general, more subjects used copy/paste than cut/paste, but few of them were entirely consistent.

The copy/paste paradigm is safer, since it anticipates the need for error recovery. This helped Subject #5, for instance, when the first attempt at modifying the inner loop was unsuccessful. Another advantage is that it can reduce the bookkeeping the programmer must do. For instance, Subject #6 used cut/paste and was driven to look back at an earlier version of the file to see what the original statements had done; this was not needed by the subjects who used copy/paste. On the other hand, cut/paste is somewhat faster, since the engineer does not have to return the original location of statements to delete them.

The tool gives the engineer the best of both copy/paste and cut/paste. In particular, it is not necessary for engineers to protect themselves against syntactic and semantic errors (although undo is still important; see Chapter 7.1). Also, the basic transformations the tool provides seem natural to these paradigms, since they are used to move syntactic units from place to place in the program, which matches the common "paste" part of both paradigms.

People make mistakes. The second conclusion is that people make mistakes, even with programs as small and as simple as this (it fits easily in a single window). Many of the individuals made minor syntactic and semantic errors that the tool guarantees to avoid. Although none of the individuals made deep "semantic" errors, this is likely due to the small size of the program.

One cost of making errors during restructuring is that it increases the time to do the restructuring. The most costly error was the syntactic one (unbalanced parentheses) made by Subject #3. The debugging process, including two unsuccessful and one successful loads, took just short of five minutes (over 20% of the total time), and accounts for this subject taking the longest of all. Although Subject #3's case was extreme, even small activities like Subject #2's deleting and reinserting parentheses to check for balance adds costs in the long run. (This subject's activity also points out that engineers are generally happy to rely on tools—in this case, the editor—to guarantee certain properties.)

Some of the errors made by the subjects would have been fixed automatically by the tool, such as the parameter mismatch by Subject #4. Others, such as the syntactic errors, would have been prevented. For others, the tool can help guide the engineer by pointing out potential problems. For instance, when Subject #2 moved the form containing `nr2` to the top-level, the tool would have returned an error indicating the problem and would have prohibited the movement until a compensating binding was inserted.

The cost of avoiding errors during restructuring cannot be ignored either (See Chapter 1.4.2). In manual restructuring, the engineer bears these costs, checking for name conflicts, preservation of value flow, updating of use sites, etc. Computer-aided restructuring fully relieves the engineer from these tedious and conceptually unimportant tasks.

Manual restructuring is haphazard. The third conclusion is that manual restructuring is haphazard compared to the computer-assisted process.

One example is the way the subjects handled the update of definitions and use sites. All but one of the subjects (#5) changed the definition and then the associated use sites. However, even the more methodical subjects (who, for instance, updated all the interfaces and then fixed all the uses) seemed to focus locally without a generally good sense of the global aspects of the process. For example, Subject #6 deleted the `nr2` binding from the main function, and then later checked to see if it had been used in that function (other than in the inner loop). Indeed it had, and the binding had to be reinserted. The same subject also separated the updates of the definition and the use by over seven minutes, about one-third of the total restructuring time. As another example, Subject #4 forgot to update the uses of changed definitions, finding out the problem when the program was executed.

One reason for some degree of complexity in the process of even the more methodical subjects might be the tendency of engineers to edit locally. That is, when an engineer is making changes in one part of the source, it is tempting to make other nearby changes, even if they are not semantically linked to the first set of changes. The temptation arises in part because a semantic change (such as generalizing the error function) is necessarily global, requiring that the function header, the function body, and all the use sites be updated. (This example only has a single use site, and the temptation to decrease editor motion would likely increase if there were multiple use sites.) This style of editing necessarily introduces some complexity and potential for error, since the engineer's mind must store a rich context for the collection of ongoing changes.

Tool support for automatically handling non-local change promises to significantly decrease this problem, since the engineer makes the change in one location and the rest are updated without added navigation through the program. By linking the modification of the use sites and of the definition site, the potential for making errors (modifying one without the other, or of updating them inconsistently) is eliminated. The tool simplifies the engineer's process from "change definition and then change all uses" to "change definition".[2] Since this new process is a subprocess of the original one, rather than an new process altogether, it is all but guaranteed to be natural to the engineer.

Another example of haphazard process is the way the subjects handled testing. Over half the subjects did not pretest the program. They later realized that they had to check the final restructured program by hand. Their tests, then, only ensured that the resulting program computed matrix multiplication, but not that there was any functional relationship between the initial and the resulting program: they only had the observer's word that the initial program in fact properly computed matrix multiplication. That is, they could not, as had been prescribed, make an effective effort to assure "that the modifications have not changed the meaning of the program in any unsatisfactory manner." A ramification of this is that the subjects were, in essence, unable to distinguish between restructuring and functional enhancement.

To paraphrase Dijkstra, testing can be used to show the absence, but not the presence, of equivalence after restructuring [Dijkstra 72]. In contrast, use of a restructuring tool that preserves meaning relieves the engineer from this concern and responsibility. Of course, the tool itself is not, of course, guaranteed to be free of errors. But as DeMillo, Lipton, and Perlis noted, confidence in a tool is in part a social process [DeMillo et al.

[2]It may be possible to define a transformation that permits the engineer to modify a use site and to use that to update the definition and the other associated use sites. The potential conceptual difficulty is in properly computing the appropriate actual parameters at the other use sites.

79]. There is no reason to believe, however, that restructuring tools cannot gain from engineers the same degree of confidence as compilers have. If and when that stage is reached, the tool will be one of the last places to look if there is an error in the restructured program. It is critical to remember, of course, that the tool does not guarantee that the restructured program is correct with respect to a higher-level specification, but only that it is equivalent to the initial version of the program.

3.3 Conclusions

The experiment was limited in several ways. The primary limitation is that even if the results validated the benefits of tool-aided over manual restructuring, this does not directly indicate that it helps address the long-term objective of reducing the costs of overall software development and maintenance. The reason for this is that the changes requested in the experiment retained functional equivalence of the program, so no direct understanding of the effect on functional changes is gained. The other limitations are that the number of subjects was small, the restructuring of only one program was studied, the experiment was not replicated, etc. Despite these limitations, however, the experiment is a useful step in evaluating whether tool-aided restructuring is worthwhile.

The first goal of the experiment was to increase understanding of how a restructuring tool in the style of the one described earlier affects process. The central lesson from the experiment is that a tool like this promises to make the restructuring process more methodical, allowing the engineer to focus more clearly on the difficult, aspects of modifying software that require human judgment. Other important lessons include the simple but useful ability to eliminate some classes of errors during restructuring and to relieve the engineer from the need to manually perform critical activities such as testing.

The second goal of the experiment was to see if the tool's model was consistent with how engineers restructure manually. The experiment did increase confidence in this dimension. Each operation applied during the tool-based restructuring is a direct analog to collections of operations that essentially all of the subjects performed.

One deviation from the model was that the subjects tended to use copy/paste and cut/paste *before* editing the text being pasted. This meant that, for example, the initial version of the inner product function had the `nr2` reference and the embedded set of `r` in it. This was then refined down to the correct form. This is slightly contrary to the approach encouraged by the tool, which prohibited this approximate-and-deviate approach to restructuring. Both the experiment and the taxonomy of structure in Chapter 2.3 imply that it will be fruitful to explore these more forgiving transformations.

In fact this approach is not contrary to the model *per se*, but is more a feature of the individual transformations, and has been implemented for `extract-function`. For `extract-function`, approximate-and-deviate for extracting the inner product without first repairing the text to be extracted would automatically add parameters for `nr2` and `r` in the interface of `inner-product`. The engineer would then refine inner-product by applying meaning-preserving transformations to remove the extra parameters. A benefit of this approach is that the structural inconsistencies are explicit as extra parameters to `inner-product`.

Even this small experiment is convincing that the basic concept of computer-assisted restructuring is a potentially valuable approach to reducing the overall costs of software evolution. Several investigations are suggested. Improving the user interface is one path to explore. Another is to attempt more experiments: repeating this same experiment on a larger sample size; using a different restructuring ; and perhaps most important, an experiment that asks subjects to make a functional change rather than simply restructure, with the intention of learning more about to what degree programmers separate the restructuring and maintenance processes.

Chapter 4

A Model for Global Program Restructuring

A critical aspect of the transformational approach to global restructuring is ensuring that a transformation preserves meaning. A model of global restructuring is needed to help reason about and implement meaning-preserving restructurings.

Intuitively, a meaning-preserving restructuring cannot affect the values that would be consumed or produced by an expression during program execution, since this would change the behavior—the meaning—of the program. There are two aspects to this. One is the production–consumption relationship of values between expressions during execution, often caused by the store of a value in a variable by one expression and the subsequent fetch of the value from the variable in another expression. The other aspect is the identity of the operations themselves—in general, an operation in the program cannot be changed since, when executed, the values produced by the new operation would be different than before. Together these two properties represent the stream, or flow, of values between operations during execution. To preserve meaning, then, the restructuring tool identifies the flows from the expression being changed and applies compensating changes to assure that the flow will be preserved. For example, when the parameters of the function `push` are swapped (Recall Figure 1.1), the need to update the calls on `push` is inferred by the flow of the procedure value to each call point.

Abstractly, such a transformation, δp, is a function from source programs to source programs, $P \xrightarrow{\delta p} P'$. However, the common computer representation of programming language syntax, the abstract syntax tree (AST), does not efficiently represent all semantic properties for preserving meaning, such as the producer–consumer relationships between operations. This is due primarily to the implicit references to variables through pointers, procedure call, and side-effects (See Chapter 1.6).

$$\begin{tikzcd}
P \arrow[r, "m"] \arrow[d, dotted, "\delta p"'] & G \arrow[d, "\delta g"] \\
P' & G' \arrow[l, "m^{-1}"']
\end{tikzcd}$$

(a)

$$\begin{tikzcd}
P \arrow[r, "m"] \arrow[d, "\delta p"'] & G \arrow[d, dotted, "\delta g"] \\
P' \arrow[r, "m"] & G'
\end{tikzcd}$$

(b)

$$\begin{tikzcd}
P \arrow[r, dotted, "m"] \arrow[d, "\delta p"', ""{name=A}] & G \arrow[d, "\delta g", ""'{name=B}] \\
P' \arrow[r, dotted, "m"] & G'
\arrow[from=A, to=B, "m"]
\end{tikzcd}$$

(c)

Figure 4.1: Diagrams of transformation (dotted arrows are implied mappings)

On the other hand, the producer–consumer relationship between operations is conveniently represented by the property of dependence [Podgurski & Clarke 90]. The Program Dependence Graph (PDG) [Kuck et al. 81][Ferrante et al. 87] explicitly represents dependences between the operations of the program. Essentially, a PDG is a graph with the vertices representing program operations, and the edges representing the flow of data and control between operations. One way the PDG could be used for transformation would be to translate the program into its PDG form, perform the transformation, and then convert it back to program form, as shown in Figure 4.1a. This allows using the nice mathematical properties of the PDG for reasoning about the correctness of the implementation of the transformation function, as well as for the efficient application of semantically-oriented algorithms.

However, the nature of the effects on the resulting program cannot be ignored. Intuitively, the PDG is a directed cyclic graph, but the AST is constrained to a tree structure: the PDG is syntactically a more general representation. Transforming a PDG in absence of constraints from the original program threatens to produce restructurings that cannot be represented as programs, or are not exactly what the engineer applying the transformation had in mind. For example, the PDG removes inessential sequential execution relationships present in the AST. Thus, when unparsed, two semantically unrelated statements can be placed arbitrarily with respect to each other, which will be confusing to the engineer. Although this problem can be reduced by constraining the ordering of unchanged code to remain unchanged,[1] the constraints on the layout of transformed code cannot be so easily stated. Likewise, the PDG collapses scopes from syntactically *permitted* relationships to *actual* semantic relationships. For example, if a program statement does not use any variables defined in its immediately enclosing scope, it may be unparsed into a location outside that scope. However, this statement originally may have been located in this scope for purposes of a future enhancement that will use its variables.

From a performance standpoint, the time required to unparse the PDG after each transformation must be considered, since the programmer cannot be expected to restructure the PDG directly. PDG unparsing is in NP, although in practice this is not considered to be a serious problem [Horwitz et al. 89]. However, even a linear algorithm (the quickest possible) may be a problem if rapid restructuring is desired.

Performance and the dependence on the original syntax of the program imply that the AST must be the core program representation that is manipulated by the programmer and the tool. Thus the requirement to not arbitrarily reorder statements is enforced by designing transformations for the AST, and the PDG is used only as a notation for showing that transformations preserve meaning and also in the tool for quickly retrieving needed dependence information. Since scoping impacts the PDG, a notation called

[1]This constraint is applied for ordering statements in program version merging [Horwitz et al. 89], and is implemented using maps between the PDG and AST (Susan Horwitz, Personal Communication, 1991) similar to those discussed in Chapter 5.3.

contours is derived from the AST to help reason about transformations in the PDG. The commutative diagram in Figure 4.1b is suggested, with the PDG G' being reconstructed from P' after δp is performed.

This approach, however, will be unacceptably time-consuming if δg is not actually performed (See Chapter 5.6). If both δp and δg exist for a particular transformation, then each representation can be updated by its own transformation procedures, yielding efficient updating of both the AST and PDG. This is visualized in Figure 4.1c, with (a now overloaded) m mapping the application of δp on program P to the isomorphic application of δg on G.

There are two practical difficulties in mapping transformations between representations. First, in meaning-preserving source transformations are global and thus difficult to reason about. This requires a way of breaking the global transformation into local parts. Second, a way of relating a change in one representation to a change in another is required. To address these needs, an equation called the *globalization equation* is defined for designing a correct AST transformation. It defines how several local AST transformations are composed—with relationships derived from the PDG—into a meaning-preserving global transformation. Also, by defining a small set of local substitution rules on the PDG that preserve meaning, it is possible to reason locally about the equivalent—but textually distributed and implicitly related—changes in the AST. In the process of applying the substitution rules to show an AST transformation preserves meaning, its corresponding PDG transformation is derived.[2]

Although the approach is not rigorous, it has been a practical aid in designing and implementing the tool and its restructuring transformations, which are described in the next chapter. Potential improvements to the model are discussed in Chapter 7.5.

[2]Although the focus here is on the AST and PDG, the Control Flow Graph (CFG) (See Chapter 5.3) underlying the PDG plays a role as well, since it represents the sequential execution relationships of the program well, which the AST and PDG do not. These are used for checking that moving expressions in the AST does not violate any relationships in the PDG. However, this role is not as significant as the AST's and PDG's roles, and the relationship between the AST and CFG is straightforward, and so the CFG is not discussed here.

4.1 The Program Dependence Graph

The PDG explicitly represents the key relationship of dependence between the operations in the program. Simple graph algorithms and set operations can extract this information. Another advantage is that the PDG has been a popular program representation for aiding program parallelization [Kuck et al. 81][Larus 89], optimization [Ferrante et al. 87], slicing [Ottenstein & Ottenstein 84][Horwitz et al. 90], and version merging [Horwitz et al. 89]. This extensive body of knowledge, combined with the right semantic support, make PDGs a good foundation for preserving meaning during restructuring.

4.1.1 Definition

A program dependence graph is a set of vertices that represent the primitive operations in the program, and a set of directed edges that connect the vertices. An edge typically represents the flow of data or control between two operations, and is called a dependence. An edge e representing the flow of data between two operations u and v is called a flow-dependence, and is denoted $e = FD(u, v)$. If the data flow dependence is due to a variable s being set in u and used in v, the edge is labeled by the variable definition carrying the flow, $e_{\mathsf{s}} = FD(u, v)_{\mathsf{s}}$. An unlabeled flow denotes direct transmission of the result of u to v without a program variable, as in Figure 4.2 where the if vertex consumes the value from the $<$ vertex. A control-dependence edge represents an on–off switch for its destination vertex. Its denotes the success or failure of the conditional operation of the predicate vertex (typically denoted by the symbol p) at its source, such as an `if` vertex in the PDG. A control dependence edge is labeled true or false to denote whether the destination vertex is activated on success or failure of the predicate vertex. Thus a true branch of predicate vertex p to v is denoted $CD(p, v)_{\mathsf{true}}$. By definition, in the program only one of the paths of control denoted by e_{true} or e_{false} can executed on evaluation of the program predicate denoted by p.

Some operation vertices generate a constant data flow-dependence (such as the value 1). In Figure 4.2 and others, to avoid cluttering they are denoted as a constant value on the flow-dependence with no source vertex. Another special class of vertices are the input and output vertices, which denote the read and write statements of the program. These are different from other vertices because the value on an output edge is not entirely dependent on the values on the input edge. To treat this property conservatively, input and output to files is treated as reading and writing a common global variable (i.e., the file system).

A PDG may also support some additional edge types: anti-dependences, def-order dependences, and output dependences. Each, for a different case, implies that the source vertex is necessarily executed before the destination vertex in the program, even though there is no explicit flow of data or control between them. The PDG representation used in the prototype [Larus 89] has anti-dependences (edges denoted by AD in Figure 4.2) and def-order dependences [Horwitz et al. 88] (denoted by DO).

An anti-dependence relates a vertex u_x retrieving from variable x to a subsequent vertex $d_{2\mathtt{x}}$ defining the same variable, if the definition can overwrite a prior definition $d_{1\mathtt{x}}$ for u_x. It is said that $d_{1\mathtt{x}}$ *reaches* u_x, and u_x *reaches* $d_{2\mathtt{x}}$. The anti-dependence $AD(u_x, d_{2\mathtt{x}})$ asserts that u_x must be executed before $d_{2\mathtt{x}}$ so that the original value stored by $d_{1\mathtt{x}}$ is not overwritten. In other words, when manipulating the source program, the expression containing the use should not be moved after the expression containing the subsequent definition. In the example above, there is an anti-dependence carried by variable `x`, from the expression `y := x` just after the conditional to the expression following it. This prevents a simple swap of these expressions. However, if all the prior `x` definitions flowing to the use of `x`—in this case the first expression in the example and the true branch of the conditional—are moved with the use, they overwrite all other definitions before they reach the use, so the move does not change the meaning.

Output dependences are analogous to anti-dependences, but for definitions reaching definitions rather than uses reaching definitions. As with anti-dependences, they are useful for checking if it is legal to move an expression. In Figure 4.2 there is an output dependence carried by variable `x` between the assignments in the initial definition `x :=`

1 and the last expression, `x := 0`. Output dependences are not directly modeled in the representation used in the prototype, but are dynamically computed as needed.

A def-order dependence asserts that two definitions of variable `x` $d_{1\mathtt{x}}$ and $d_{2\mathtt{x}}$ both have a flow-dependence to the same use $u_{\mathtt{x}}$, but that *if* $d_{2\mathtt{x}}$ is executed it overwrites (*kills*) $d_{1\mathtt{x}}$. This implies that the two definitions cannot be under the same control dependence predecessor, such as the opposite arms of the same conditional expression. Def-order dependence is like output dependence, but it excludes unconditional kills of definitions [Horwitz et al. 88]. In Figure 4.2 there is a def-order dependence from the initial definition `x := 1` to the definition of `x := 2` in the `if-then` clause, since (1) they both can reach the use of `x`, `y := x`, directly after the conditional, but (2) the second definition conditionally overwrites the first definition.

It is useful to know whether a kill is conditional because an unconditionally killed dependence has *no* semantic relation to subsequent variable uses, while a conditionally killed definition shares a use with the killing definition. In other words, it is not possible to syntactically separate one definition's uses from the other's. For example, reversing the order of two def-order definitions will result in a different value overwriting the first definition before being accessed by the shared use, which cannot preserve the behavior of the program. On the other hand, reversing two output dependent expressions (that are not def-order dependent) will not affect the meaning of the program if all of the uses of the second definition are moved before the first definition, and other dependences are preserved.

A dependence can be further classified as *loop-carried* [Horwitz et al. 88], meaning that the dependence arises only if a loop body is iterated more than once. This classification determines if a two vertices are dependent on the first execution of a loop (a normal dependence), or in subsequent iterations (loop carried).

Procedures are treated as individual PDGs that are linked by calls. Unconditional control flow dependences are used for the transfer of control, and data flow dependences represent the passing of call arguments and return values. This is similar to the System

```
local x := 1
begin
  local y
  if x < 5 then
    x := 2
  end
y := x + 1
```

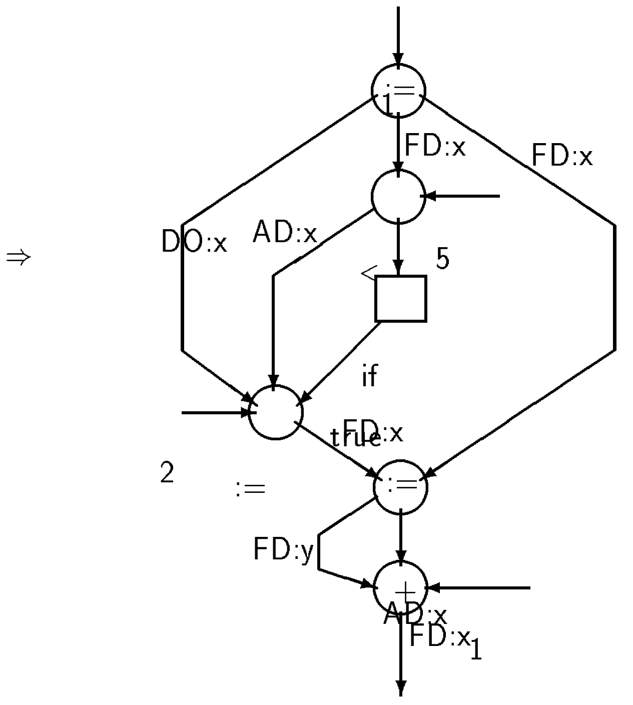


Figure 4.2: A program and its program dependence graph

Dependence Graph representation [Horwitz et al. 90], but does not remove spurious transitive dependences due to call sites sharing the same entry and exit to a procedure's PDG. The presence of pointers in Scheme makes the analysis to remove spurious dependences difficult.[3]

4.1.2 Additional Terminology

To describe some semantic relationships between vertices in the PDG requires some additional notation (taken in part from an investigation on computations on def-use graphs [Podgurski & Clarke 90]). This notation will also be used to describe changes to the PDG.

[3]S. Horwitz, Personal Communication, 1990.

For two vertices u and v, $u \overset{op}{=} v$ means that they contain the same operation. When a vertex is *executed*, actually the execution of the corresponding operation in the program is implied. Likewise for other operational terms applied to the PDG. The *behavior* or *meaning* of a PDG is the behavior or meaning of the program that it denotes. In particular two PDGs have the same meaning if their respective programs have identical input–output pairs for all defined inputs.[4] Stated slightly more optimistically, a PDG G' derived from G *preserves the meaning* of G if G' has identical behavior for all defined inputs for G. This allows G' to be defined on inputs for which G is not.

If there is a dependence edge $e = (u, v)$, then v is a *successor* of u, and u a *predecessor* of v. A traversal in a PDG from v_1 to v_n via dependence edges can be described by an ordered list of the visited vertices called a *walk*, $W = v_1 v_2 ... v_n$. By using a variable subscript on a vertex or walk symbol, several walks are described concisely. For example $W_i = uv_i$ describes all the walks from u to its immediate successors. The v_i also may be listed explicitly with braces: $W_i = u\{v_1, v_2, ... v_n\}$. If two vertices, u and v have the same variable subscript, in a walk description, the variables are assumed to be the same. Thus walk $x_i y_i$ denotes the walks $\{x_1 y_1, x_2 y_2, ... x_n y_n\}$. The notation p^* denotes that p is a successor of itself, and denotes all walks with zero or more visits to p.

Walks extend to edge visits as well. Given vertex walk $W = w_0 w_1 \ldots w_n$, then the edge walk $E = e_1 e_2 \ldots e_n$, demonstrates W, given that $\forall w_i$, $0 < i \leq n$, $\exists e_i = (w_{i-1}, w_i)$. Edge and vertex walks may be interleaved to describe the exact path traversed on a walk. Thus $W = w_0 e_1 w_1 e_2 \ldots w_n$ is a traversal of the graph such that $e_1 = (w_0, w_1)$, and so forth.

Informally, a vertex u is semantically dependent on v if for some input to v, the value of u is influenced by v. The PDG captures conservative versions of semantic dependence. Two vertices x and y are directly dependent if there is a control or data flow-dependence edge (x, y) in the dependence graph. Vertices x and y are *flow dependent* if there is a

[4]This precludes ascribing meaning to the running time of a program. Although running time is a critical semantic property of real-time systems, this complication has not been considered at this stage of the research.

graph walk W such that xWy, but including only data and control flow-dependences. The other edges are excluded because they do not represent the flow of values or control in the execution of the program.

For the design of the PDG used in this thesis [Larus 89], flow dependence is the same as *strong syntactic dependence* (SSD) [Podgurski & Clarke 90]. If a vertex u is SSD on v, then u may be (is likely to be) semantically dependent on v. However, there is also a case in which a semantic dependence is not SSD. In particular, if a loop or recursion does not terminate, then the code after the loop (in the sequential execution of the program) is never executed. So although the loop may not have any visible control or data flow dependences to this subsequent code, it still affects it. This *weak* control dependence combined with SSD is called *weak syntactic dependence* (WSD) [Podgurski & Clarke 90]. WSDs are not checked in the tool transformations because changes to termination properties are avoided.[5] Flow dependence is of interest, for example, when considering a mutation to an expression. Walking the flow-dependences from the mutated element reaches all the (strongly) affected parts of the program [Podgurski & Clarke 90]. Walking flow dependences backwards from a component yields its (strong) slice [Weiser 84][Horwitz et al. 90][Podgurski & Clarke 90].

4.2 Contours

Contours are an abstraction of the essential semantic properties that the AST represents in an efficient and complete form, but the PDG does not. The PDG does not represent all the syntactic properties of a program, and these are important for successful source-to-source restructuring. For example, changing the PDG to reference a

[5] Although transformations cannot change the termination properties of a program, a transformation may change the output behavior of a non-terminating program by moving an expression from before a non-terminating expression to after it, or *vice versa*. Theoretically, however, the meaning of the program is undefined in either case, so the meaning is not changed [Hoare et al. 87]. In the case that the program is not supposed to terminate, such as an operating system, the infinite loop has a visible external behavior (I/O, that is), so an expression cannot be moved over this loop if it will have any impact on the program's behavior (such as changing the order of I/O operations). In any event, if it was desirable to handle WSD the dependences could be appropriately extended.

new variable may preserve meaning in the PDG, but not in the program if the new variable has the same name as another variable in the same scope. Most previous applications of PDGs have not required preserving scope properties because they are either non-manipulative [Weiser 84] or the *source* is not being manipulated, such as in program optimization [Ferrante et al. 87] and parallelization [Larus 89]. Program version merging—a less constrained domain than program restructuring, since the meaning of the merged program will usually be different from the originals—uses unparse techniques to rederive the source representation, but it can fail because the merged PDG cannot always be unparsed into a legal program [Horwitz et al. 89].

Scoping is the most semantically critical syntactic property. Other syntactic properties are important, too, but are much more static or are captured more readily in the PDG. As an example of a static syntactic constraint, a typical programming language like Scheme does not support return of an expression's result to more than one expression. The destination is determined by the child-to-parent relationship, and an expression has only one immediate parent. The (illegal) change in the PDG that could imply such an illegal structure—modifying a vertex to have two unlabeled successor data flow-dependences—is simple. However, the syntactic constraint is also easy to check and enforce in the PDG; a check of the successor edges of a modified vertex is sufficient. On the other hand, in the PDG it is not so easy to prohibit a change that yields two variables of the same name in a scope, since storage is abstracted away as the direct communication of values between operations by data flow-dependences.

By adding scoping constraints to manipulations of the PDG, preserving the meaning of the program can be assured. The AST, the common hierarchical representation for a program in programming tools, readily represents scope structure. This scope structure can be abstracted as a concept called *contours*, which suggests the hierarchical stacking relationships of nested scopes. Given contours and a way to relate components in the AST and PDG, the two representations can be combined into a single formalism to reason effectively about both flow-dependence and scope structure. Together these enable reasoning about moving an expression with regard to both the reordering of expression evaluation, explicit in the PDG, and its ability to access the variables it needs, which is revealed by contours.

This relationship between a scope binding and the references on that binding is called *scope dependence*. As with the other kinds of dependence, it is important to preserve scope dependence during transformation. In particular, a data flow-dependence carried by a variable cannot be preserved if the variable's scope dependence is not. Hence, a data flow-dependence is called *consistent* if it obeys scope dependence.

What a scope contains is defined by the hierarchical relationships in the program text, which is captured precisely by the AST. For example, in Figure 4.3 the `local y` declaration in the program text is visible to only the expressions following it in the same `begin–end` declaration. Without loss of generality, assume that there is only one `local y` declaration per `begin–end` declaration. Global declarations represent the outermost scope of the program, and are declared at the root of the AST.

```
local x := 1
begin
  local y
  if x < 5 then
    x := 2
  y := x
  x := 0
end
```

$\Rightarrow$

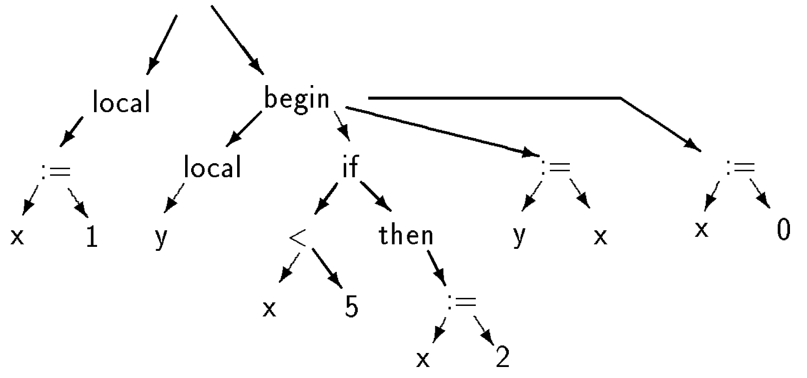


Figure 4.3: A program and its abstract syntax tree

In the AST for the program on the right, this is captured by the parent and child relations in the program. In particular, the y binding cannot be visible to an expression unless a single step up, followed by any number of steps down, can reach the expression. This is captured in the following:

Definition 1 (Containing Contour) *For expression e and variable binding declaration $b_{\mathtt{x}}$ in an AST, if*

$$e \in children^{*}(parent(b_{\mathtt{x}}))$$

then contour $b_{\mathtt{x}}$ contains *e, or $contains_{ast}(b_{\mathtt{x}}, e)$.*

Containment in block-structured languages is not sufficient for a reference to a variable named `x` in e to actually reference $b_{\mathtt{x}}$. If there is some other declaration of `x` containing e but closer than $b_{\mathtt{x}}$, then this other declaration is the (unique) defining binding for e. More formally,

Definition 2 (Defining Contour) *For AST binding $b_{\mathtt{x}}$ that contains expression e, $b_{\mathtt{x}}$* defines `x` *for e, or $e \in_{ast} b_{\mathtt{x}}$, if $b_{\mathtt{x}}$ does not contain any $b_{\mathtt{x}}{}'$ that contains e.*

Intuitively, a contour that defines a variable for an expression supplies the storage for that variable. In Figure 4.3 the storage for the assignment to the `y` reference just after the conditional is supplied by the `local y` declaration above it, since according to the definitions `local y` defines `y` for that assignment.

For PDGs vertices to be related to contours, they must inherit the notion of location (i.e., parent, child, and sibling syntactic relationships) from AST vertices. Locations are assigned to PDG vertices by mapping them to the AST through the one–one correspondence between primitive operations in the AST and the PDG. This relation is defined by the functions m and m^{-1}, which map a vertex in the AST to a vertex in the PDG and *vice versa.* Since PDGs lack variable bindings, the PDG vertex that corresponds to the initialization (i.e., the first set) of a variable is used as the representative binding declaration. Thus, $m(b_{\mathtt{x}}) = d_{\mathtt{x}}$, where $d_{\mathtt{x}}$ is the first set of variable x. So given $d_{\mathtt{x}}$, and another PDG vertex v, their contour containment and membership operations are defined in terms of the AST definitions:

$$\begin{aligned} contains_{pdg}(d_{\mathtt{x}}, v) &= contains_{ast}(m^{-1}(d_{\mathtt{x}}), m^{-1}(v)) \\ v \in_{pdg} d_{\mathtt{x}} &= m^{-1}(v) \in_{ast} m^{-1}(d_{\mathtt{x}}) \end{aligned}$$

Now it is possible to check whether a dependence edge is consistent with the contours of the program. An inconsistency occurs if a vertex is moved out of the contour that defines one of its variable references. To check this requires verifying the following property for each predecessor and successor edge of the moved vertex:[6]

Definition 3 (Contour-Consistent Dependence) *PDG edge* $e_{\mathtt{x}} = (u, v)$ *is* consistent *if* $u, v \in b_{\mathtt{x}}$.

Figure 4.4 below superimposes the containment contours for the scopes of `x` and `y` in the program. The vertices within the box labeled `y` are contained in the `y` contour. Likewise, the vertices within the box labeled `x` are contained in the `x` contour. This includes the vertices within the `y` contour. However, if there were another `x` contour within `y`'s—due to a scope declaration within the scope of `y`—the vertices within *that* contour would not be members of—defined by—the outer `x` contour.

The key relation is the defining contour, but when *moving* vertices the containment relation remains important because its default properties (determined by nesting) must be used to recompute defining contours. Thus the containing relation must be available as the primitive contour relation.

4.3 Meaning-Preserving Graph Transformation Rules

Since PDGs locally and explicitly represent dependence, and contours are relatively easy to impose upon them, the PDG is used as the primary notation for reasoning about the correct design of meaning-preserving transformations. Meaning-preserving changes to the PDG are described by and constrained to a small set of substitution rules that

[6]This check does not catch an inconsistency if a vertex sets a variable that is never used, but to catch this case the check can be generalized to examine vertices.

```
local x := 1
begin
  local y
  if x < 5 then
    x := 2
  y := x
  x := y + 1
end
```

⇒

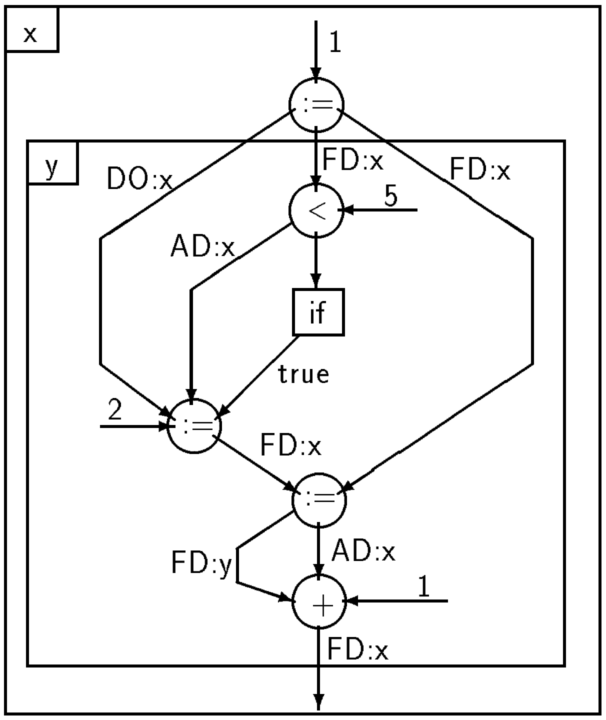


Figure 4.4: A program and its PDG overlaid with contours

preserve flow dependence and scope dependence, and do not change the operations in value-changing vertices. Preserving dependence between two vertices preserves the semantic link between them. Not changing the mutating operations preserves the actual values passed along the preserved dependences. This approach does leave the opportunity to make changes to the graph that preserve meaning—and restructure the program. For example, it is possible to add an assignment vertex that transitively transmits values without changing them. Such a change, for instance, is the first step to allowing inlined text to be extracted into a procedure: its parameters are passed by a transitive assignment.

This notion of equivalent meaning is rigorously applied by the Sequence-Congruence algorithm of Wuu Yang [Yang 90], which computes equivalence classes of equivalent programs or subprograms. Members of an equivalence class, when given identical inputs, terminate with their visible variables in the same state. Although Yang's definition of the Sequence-Congruence algorithm uses a variant of the PDG called a Program Representation Graph (PRG), it still applies to PDGs when appropriately modified. The key difference between the PDG and PRG is that the PRG uses normalized variables in the style of static single assignment (SSA) form [Cytron et al. 88].

Yang's algorithm determines the equivalence of two PDG components based on three properties, (1) the equivalence of their operators, (2) the equivalence of their inputs, (3) the equivalence of the predicates controlling their evaluation. The algorithm likewise proceeds in three steps: partitioning the vertices into sets of equivalent operators, refining the partition with respect to data dependences, and refining the partition again with control dependences. All vertices in a partition have the same behavior. A valuable subcase of sequence-congruence is *data-congruence*, which is the property that two vertices in the same partition have after just phases (1) and (2) are completed. The algorithm can be extended by using a special rewrite pass to handle the transitivity of assigning one variable to another [Yang 90, p. 62]. The extension is necessary because the basic algorithm does not recognize assignment as an operation.

By definition, subgraphs of a PDG may be modified by the substitution of sequence-congruent vertices without changing a PDG's meaning. For example, a vertex and its incoming edges can be replicated (by definition being sequence-congruent), and the outgoing edges split between the two copies [Allen & Cocke 72]. Thus PDGs can be transformed by performing replacement of sequence-congruent vertices in the PDG. This is the technique applied in this thesis for restructuring transformations. The replacements used are described below by a small set of PDG subgraph substitution rules motivated by sequence-congruence. As described above, source-to-source transformation is the ultimate goal, so a group of substitutions are allowed only if they can be correctly mapped to a syntactically legal program.

First presented are the rules that are implied by sequence-congruence, and then two for scope contours. Together these rules describe legal changes to a PDG, constrained by contour consistency so that they can be mapped back to the AST. The changes described by a rule must be the only changes to the PDG, otherwise meaning is not preserved. Similarly, any newly defined edge label (variable) is assumed to be unique, and so cannot conflict with any existing label, although the extension rule (Section 4.3.6) defines more general constraints for picking the name for a variable. More generally, the application of the subgraph substitutions must not violate the semantic contour rules. Finally, unless otherwise noted, the substitutions work in both directions; the equations describe the precise relationships.

4.3.1 Transitivity

A variable `s` that is assigned the value of another variable, `r`, has the same value, and hence the same meaning as `r`. More formally, if there is a flow dependence $e_{\mathtt{r}} = FD(u, x)$ and $e_{\mathtt{s}i} = FD(x, v_i)$, where x is an identity vertex (i.e., an assignment), then there is an indirect flow dependence from u to the v_i for which the value of the flow is not changed through x. Thus if x is removed and its input flow is spliced to its output flows: $e_{\mathtt{s}i} = FD(u, v_i)$, the meaning of the graph is unchanged. The deletion may likewise be inverted, inserting x between u and the v_i, via variable `r`.

Rule 1 (Transitivity) *Given all walks $ue_{\mathtt{s}i}v_i$ for fixed vertex u and variable `s`, identity vertex x (with the same control dependence edge as u) may be inserted between them for all or one of the v_i or more precisely:*

$$ue_{\mathtt{s}i}v_i \;\equiv\; ue_{\mathtt{r}}xe_{\mathtt{s}i}v_i$$

or for a v_a in the v_i:

$$ue_{\mathtt{s}a}v_a \;\equiv\; ue_{\mathtt{s}a}xe_{\mathtt{r}}v_a,\;\; \nexists we'_{\mathtt{s}}v_a$$

Note that these must be the *only* changes to the graph. The first case states the legal substitution of a transitive variable before the existing variable, the other case, after. The latter case is more complicated because a particular *use* of `s` is chosen for substitution (indicated by the subscript constant a), and the use cannot be using any other definitions for the same input (any other edges e'_s to v_a). To see why this constraint is necessary, which appears in the next two rules as well, consider applying `var-to-expr` on the initial assignment in the program in Figure 4.5. This implies redirecting the constant 1 flow to the two calls on `f`. However, the second call depends *also* on the assignment of `x` in the conditional, which is not being changed, so inlining the 1 in the call to `f` would not preserve meaning.

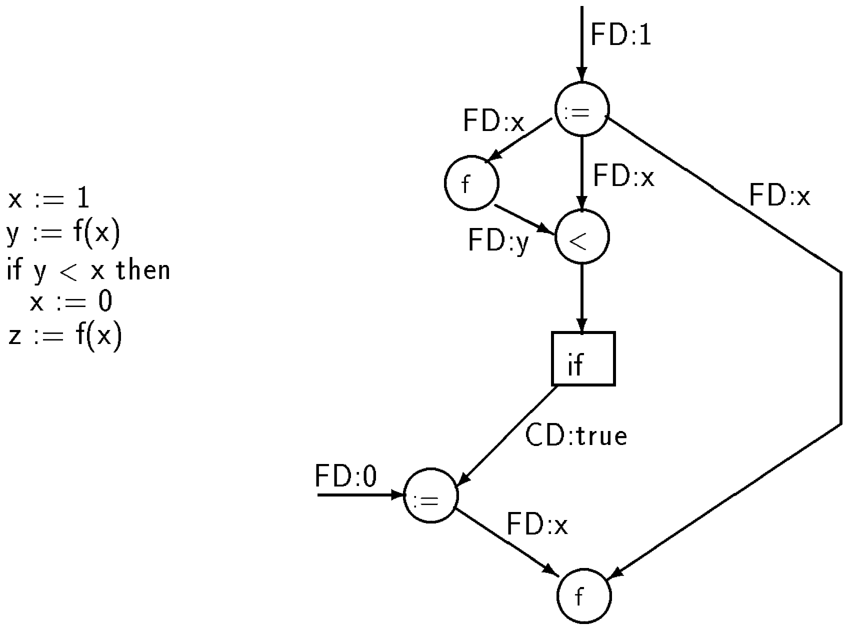


Figure 4.5: Multiple definitions can prevent transformation

In the matrix-multiply example (See Chapter 2.6), the second case of transitivity allows the constant `'matrix-multiply` to be abstracted from function `match-error` through the parameter `header` using `expr-to-binding` (See Figure 4.6). The inserted assignment vertex in the PDG is implemented with parameter passing in the program text.

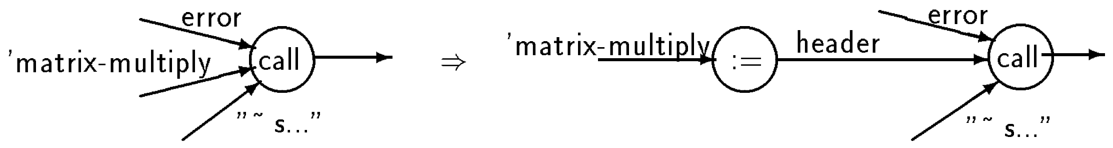


Figure 4.6: Transitivity justifies `expr-to-binding`

4.3.2 Indirection

Suppose a scalar is converted into some nested structure, such as an array or record structure. What once was a direct reference to the variable now requires dereferencing to store and retrieve values if meaning is to be preserved. This kind of change is captured by the indirection rule.

Rule 2 (Indirection) *Given all walks uv_i for fixed u, insertion of vertex n, a nesting operation, between u and v_i (with the same control dependence as u) requires inserting its inverses n_i^{-1} between n and the v_i (with the same control dependence as v_i), or*

$$ue_{\mathbf{s}i}v_i \equiv une_{\mathbf{s}i}n_i^{-1}v_i, \quad \nexists we'_{\mathbf{s}}v_a \; for \; v_a \; a \; v_i$$

This transformation allows changing the structure of variables (along with the way they are accessed) to create new data types and abstractions. Figure 4.7 shows how the data flow dependences are changed by `listify` in the example in Chapter 2.5. These changes are justified by the indirection rule.

4.3.3 Distributivity

In the PDG, if vertices u and u' are sequence-congruent, then any successor flow dependence edge of u can be made a dependence out of u'.

Rule 3 (Distributivity) *Given all data flow-dependence walks through vertex u ($x_kue_{\mathbf{s}i}v_i$), vertex u', sequence-congruent to u, may assume the flow-dependence successor $e_{\mathbf{s}}$ of u for any of the v_i:*

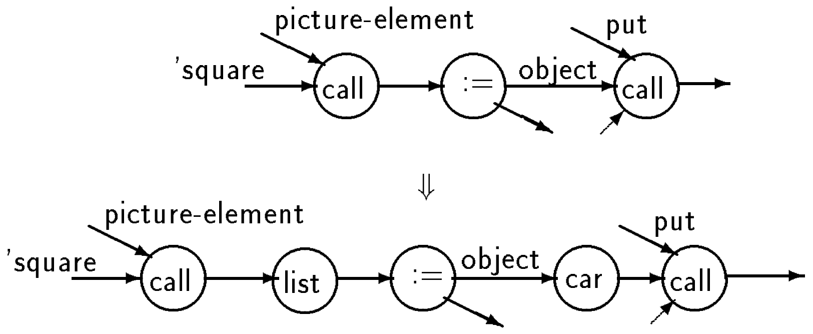


Figure 4.7: The indirection rule is used in `listify`

$$x_k u e_{\mathtt{s}} v \quad \equiv \quad x_k u' e_{\mathtt{r}} v, \quad u' \overset{op}{=} u \ \wedge \ \nexists w e'_{\mathtt{s}} v$$

Multiple edges can be moved from u to u' by applying the rule to other successors of the $e_{\mathtt{s}_i}$. The rule naturally extends to groups of vertices by applying the rule to the x_k. The vertex-equality part of the rule can be derived constructively by copying u and its incoming edges to create a trivially sequence-congruent u'. However, because replicating a vertex is essentially multiple evaluation of the vertex, it may redefine the output variable. This is why the rule changes the variable from `s` to `r`.

Figure 4.8 shows how applying `var-to-expr` to the second use of `nr2` in the matrix multiply example (See Chapter 2.6) changed the PDG, which is justified by the distributivity rule. In this case the call node is u, `nr2` is `s`, the bottom assignment is v, and `len` is the newly introduced variable `r` to carry the moved dependence.

4.3.4 Control

As with communicating data through transitive assignments, control can be successively propagated across control edges if the sense of the condition is not changed. Informally,

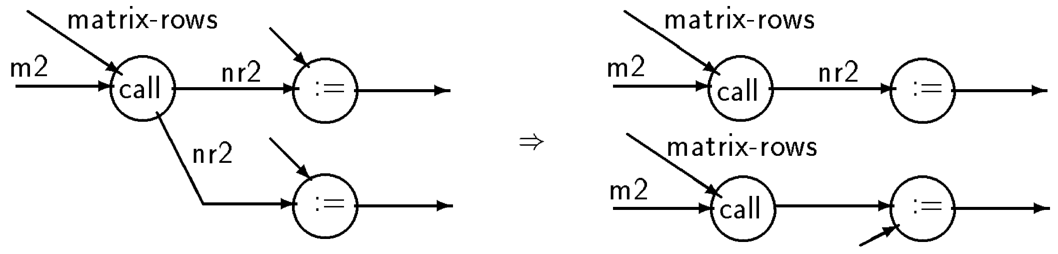


Figure 4.8: The distributivity rule is used in `var-to-expr`

if vertex x and predicate vertex p_1 are control flow successors of predicate vertex p_0, x can be made a control successor of p_1 if:

1. x is duplicated and put under the opposite control successor edge, or
2. all of x's flow successors are already under the control of the same conditional successor.

The second case asserts that if x is not executed, it will not be missed. This holds for loops, too, but only if there will be no definitions in the loop killing x's definition or the definitions that itself x uses. Trivially, if x represents a constant, a control dependence is not necessary to regulate its execution since it can neither crash the program nor change the state. However, unlike the other rules, not all cases of this rule are invertible, since loosening a control-dependence on x may allow it to be executed when its value is undefined. The constraints are stated precisely in the rules below.

When interpreting the walks in the rules, recall that control and data flow-dependences are being interleaved, and that the data flow successors u_i of a vertex x are not necessarily under the same control dependence as x, unless explicitly stated.

Rule 4.1 (Control Distributivity) *Given data flow-dependence walk xu_i and* if *vertex p_1, both successors to p_0, x can be made a control successor of p_1 by the following relation:*

$$p_0\{p_1, xu_i\} \;\equiv\; p_0p_1\{e_{\mathsf{true}}xu_i,\; p_1e_{\mathsf{false}}x'u_i\},\;\; x' \text{ data congruent to } x$$

This transformation is basically an algebraic distributivity law for control. It is also the control analogy to the distributivity rule, but since the control paths are exclusive, all the edges must be *copied* for x' and there is no need to change the labels on the edges between $x'u_i$. The figure below shows the effect of applying this rule to a PDG.

Rule 4.2 (Asymmetric Control Distributivity) *Given data flow-dependence walk xu_i and* if *vertex p_1, both with control predecessor p_0, x can be made a control successor of p_1 by the following relation:*

$$p_0\{x, p_1e\}u_i \;\Rightarrow\; p_0p_1\{ex, e\}u_i$$

This allows "distributing" a vertex into just one arm of a conditional in the case that all those dependent on it are in that arm of the conditional. Note that this can inadvertently *improve* the non-termination properties of the program (a change in WSD that reduces the set of inputs that will cause the program to not terminate) by causing x *not* to be evaluated in the case that p_0 evaluates true and p_1 evaluates false. Since performing the substitution right-to-left potentially has the opposite effect of increasing the chances of non-termination, this substitution is defined only left to right.[7]

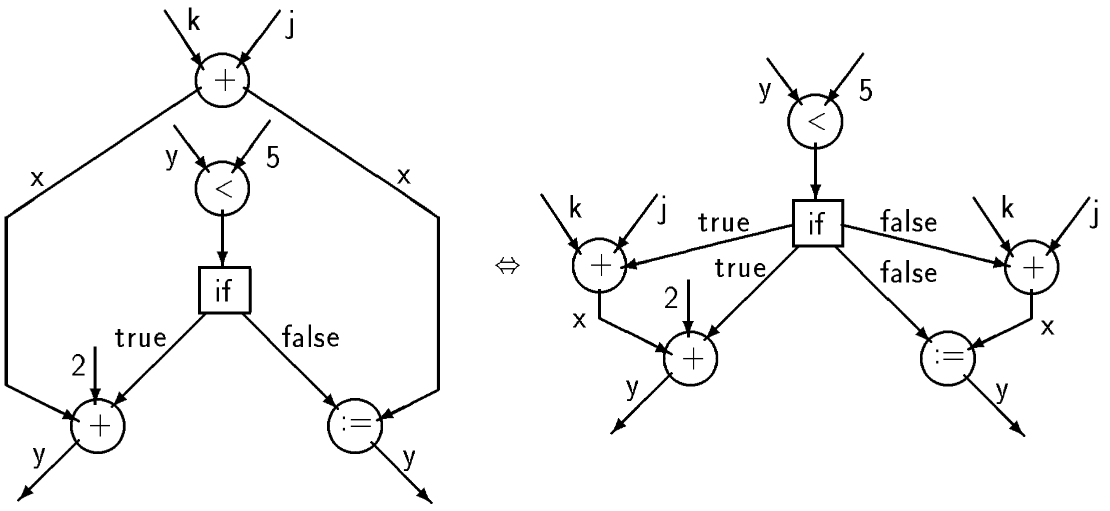
This rule may be similarly applied for p_1 that is the head of a loop (a while vertex), with extra conditions to handle the possible effects on x by iterating the loop body:

Rule 4.3 (Recursive Asymmetric Control Distributivity) *Given the data flow-dependence walks xu_i and* while *vertex p_1, both with control predecessor p_0, x can be made a control successor of p_1 by the following relation:*

$$p_0\{xe_{\mathsf{s}}, p_1{}^*e_{\mathsf{true}}\}u_i \;\Rightarrow\; p_0p_1{}^*\{e_{\mathsf{true}}xe_{\mathsf{s}}, e_{\mathsf{true}}\}u_i,$$
$$\text{such that } \nexists\; we'_{\mathrm{LC}(p_1):\mathsf{s}}u_a \;\vee\; xe_{\mathrm{AD}:s}x \;\vee\; p_1{}^*p_i{}^*ve_{\mathrm{AD}:var}x$$

[7]This rule can be generalized to allow right-to-left substitution when the operation in x can be guaranteed to terminate, but this is not described easily with sequence-congruence.

The conditional clause assures that the repeated evaluations of x will not override other variable definitions flowing to the u_i in the loop, and that the data flow-dependence walks to x are not changed. Specifically, edge $e'_{\text{LC}(p_1):\mathtt{s}}$ denotes a loop-carried (LC) dependence due to loop-head p_1 for variable `s` to any of the u_i. x's dependences would override such an edge if x were moved under p_1. Walk $p_1{}^*p_i{}^*ve_{\text{AD}:var}x$ denotes an anti-dependence to x caused by a variable definition anywhere within the loop. Such an anti-dependence indicates that a (loop-carried) data flow-dependence would be introduced if x were moved under p_1.



```
x := k + j
if y < 5 then
  y := x + 2
else
  y := x
```

```
if y < 5 then
  x := k + j
  y := x + 2
else
  x := k + j
  y := x
```

Figure 4.9: An example of the first case of the control rule

4.3.5 Substitution

There are two rules for manipulating contours and constraining the contour manipulations described by the previous rules. These follow directly from the definition of contours and the requirement that all manipulations of vertices and contours be contour consistent.

To change the name of a variable in a program requires changing the binding declaration and all of the references to that binding, and then verifying that no name conflicts have been introduced. In the PDG this requires changing the name of the references to the variable defined by the contour (as implied by the binding declaration), and then verifying that there are no inconsistent dependences.

Rule 5 (Substitution) *Contour $C_{\mathtt{s}}$ can be renamed $C_{\mathtt{r}}$ if:*

$$\forall u, v \in C_{\mathtt{s}}, \quad e_{\mathtt{s}} = FD(u,v) \quad \equiv \quad u, v \in C_{\mathtt{r}} \quad \wedge \quad e_{\mathtt{r}} = FD(u,v)$$

In the transit example, this rule permitted renaming the variable `miles-traveled` as `miles-rolled`. Figure 4.10 shows an example of the rule *prohibiting* a name change, because the new, inner `y` contour defines `y` for only one of two vertices with a `y` edge between them.

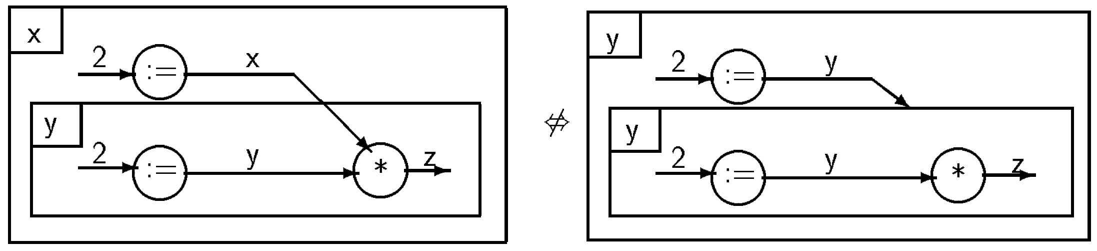


Figure 4.10: Changing x to y is *not* meaning-preserving

4.3.6 Extension

Adding a new contour $C_{\mathtt{x}}$ or changing the range of $C_{\mathtt{x}}$ does not affect the meaning of the PDG if in the result there would be no edge entering or leaving $C_{\mathtt{x}}$ with label `x`. Assuring this is slightly complicated by nested scoping rules, which requires manipulating the *contains* relation rather than the *defines* relation. This means that an additional check is required to assure that consistency is maintained.

Rule 6 (Extension) *Contour $C_{\mathtt{x}}$* containing *preexisting PDG vertices u preserves meaning if, $\forall e_{\mathtt{x}}$ such that $\exists v$ satisfying $e_{\mathtt{x}} = FD(u,v) \vee FD(v,u)$, then $e_{\mathtt{x}}$ is a contour consistent dependence.*

Contraction, the inverse of extension, is captured by the rule as well. Also note that this handles moving a vertex in and out of a contour, since these are just extension and contraction operations. In the transit example, the extension rule permitted importing `total-miles` into the transit module. Figure 4.11 is an example that requires extension to add the contour for `len`, which is needed for the substitution rule when applying `expr-to-var` to the second use of `nr2`.[8] The extension rule holds here because the new contour defines `len` for the two vertices connected by an edge labeled `len`, and does not block the incoming edges labeled `m2` and `matrix-rows`.

4.4 Defining a Meaning-Preserving Transformation

Now that a set of meaning-preserving substitution rules for the PDG are defined, it is possible to describe how these are applied in defining a *program* transformation that preserves meaning. This transformation relationship between a program and its PDG is defined using a commutative diagram to relate manipulations on the graph to manipulations on the program.

[8] `expr-to-var` at first appears to require the transitivity rule when looking at the change to the program. But the graph shows that it is only necessary to add a label to the unlabeled edge.

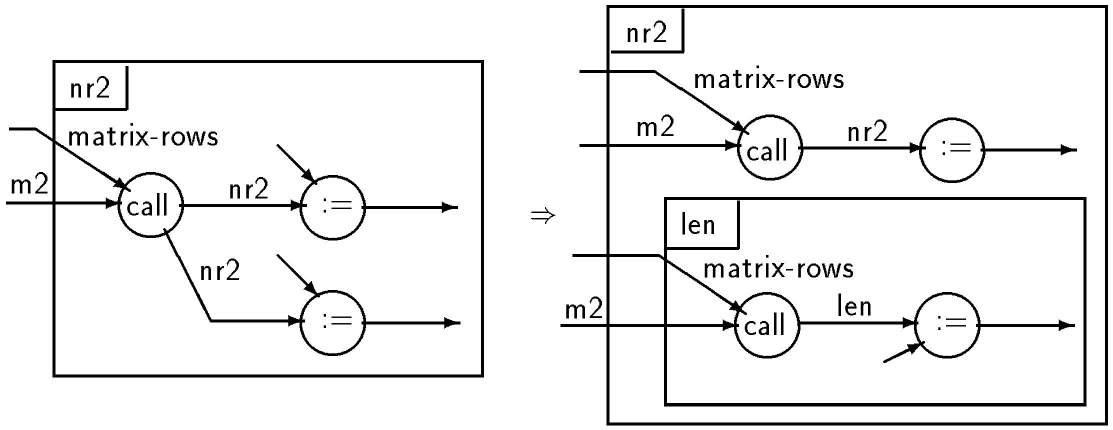


Figure 4.11: Extension is used to add the contour for `len`

Given program P and PDG G such that $m(P) = G$, then transformation $\delta p\colon P \rightarrow P'$ preserves the meaning of P if $\delta g : G \rightarrow G'$ is a composition of meaning-preserving substitution rules such that $m(P') = G'$:

$$\begin{tikzcd}
P \arrow[r, "m"] \arrow[d, "\delta p"'] & G \arrow[d, "\delta g"] \\
P' \arrow[r, "m"] & G'
\end{tikzcd}$$

Figure 4.12: The commutative diagram for program and PDG transformation

This relation is used to prove that a tool transformation (δp in the diagram) *a priori* preserves meaning by showing that $\delta p = m^{-1} \circ \delta g \circ m$, or, if we define m so that it operates on functions as well as data, $m(\delta p) = \delta g$. This proof of correspondence is performed manually, and currently informally, for designing tool transformations. By performing this proof off-line, an expensive run-time proof that G' is equivalent to G is avoided.

And once given δp, it is not necessary to apply m^{-1} to reconstruct the program from the PDG.

4.4.1 Globalization

To demonstrate the relationship $m(\delta p) = \delta g$ requires an understanding of how a change to the PDG maps to changes in its corresponding program. This knowledge is applied by examining a program transformation procedure and mapping its change operation into an equivalent set of substitutions in the PDG, thus showing the program transformation preserves meaning. Since PDG substitutions are local, but program transformations are global, this mapping is called *globalization*.

The commutative diagram is a guide to the mapping. Given the above transformation rules, and mappings m and m^{-1}, it seems it that should be trivial to construct a program transformation. However, the known algorithms for m and m^{-1} are batch, data-oriented algorithms. Although m might only need to be performed once if the program is repeatedly reconstructed from the PDG, repeatedly performing m^{-1} with large programs cannot achieve good enough performance for interactive transformation. Instead, it is desirable to *lift* m and m^{-1} so that they operate on functions δg and δp instead of the data passed to and from δg or δp. The result would be a direct incremental manipulation of the program, rather than batch reconstruction.

However, the formal basis for such a lift of program–PDG mappings is in its infancy [Cartwright & Felleisen 89]. So rather than lifting m and m^{-1} using a formal basis, the technique is informally applied in the processing of implementing a transformation, based on the transformation builder's knowledge of the programming language and PDGs. The framework for deriving the correspondence δg and δp is the *globalization* equation, which helps to correctly define the complete meaning preserving transformation δp given the engineer's change L to an expression e in P. The order of composition of the changes is not important since each use is in a syntactically independent location:

$$\delta p = \left(\prod_{u_i \in m^{-1} \circ fs \circ m(e)} C(u_i) \right) \circ L(e)$$

where *fs* is a function retrieving the flow-successors of a PDG vertex, and C is the compensating operation applied to the u_i, the uses of e. Note that the data-oriented m and m^{-1} functions are still present, but limited to retrieving program components based on search in the PDG. This retrieval can be performed quickly without actual PDG or program building operations (See Chapter 5.6). The L and C program transformations are chosen by the transformation builder so that they map to the substitution rules (and also that they make sense to the tool user). The result is that the core of the substitution rules have been successfully mapped to the program, with the remaining accesses to the PDG being efficiently supportable.

The operation $m^{-1} \circ fs \circ m(e)$ may be denoted as $uses(e)$. As used in this context, the uses are the program representation of the variable uses denoted in the def-use graph.

Now consider using the globalization equation to design a transformation. The compensating transformation C is known statically for a given L. For example, the engineer swapping formal parameters directly implies swapping call arguments. But when the transformation is applied, the actual calls to be updated are found by mapping e through the PDG using m. These properties are reflected in the equation by the fact that C is a constant to δp, but e is a parameter. Likewise, the semantic checks are mapped to the PDG and their results converted back to program syntax form for failure reporting. To construct a transformation requires, then, the program, P, with a mapping to G via m, and a set of transformations δp defined by C and L. The arguments for C and the checks are computed using m and traversal in the PDG.

The program transformation is proven correct by showing that the $C(u_i)$ and $L(e)$ map to the PDG as a composition of the substitution rules, δg. When performing this mapping the transformation builder must show that the rules are *sufficient* for the program transformation to preserve meaning. This requires showing that (1) every possible edge from the locally-changed object has been handled, (2) none of its input edges is affected, (3) that the compensations do not adversely affect any other incoming edges in the compensated objects, (4) nor any of their output edges. No other edges have to be checked because if the incoming and outgoing edges to the modified subgraph are unchanged in meaning, then the rest of the PDG cannot have been affected.

This is what `var-to-expr` (when given a variable definition to inline) looks like abstractly, where $(\mathtt{v}, e)$ is the expression that defines `v`:

$$\delta p = \left(\prod_{u \in uses((\mathtt{v},e))} substitute_{(e,u)} \right) \circ remove_{(\mathtt{v},e)}$$

Of course it remains to be justified with substitution rules that this is meaning-preserving, and the substitution rules imply checks that are lacking. The nature of these checks is discussed next; then the justification follows.

4.4.2 Preconditions for Transformation

The substitution rules are conditional rules, constrained by the scope rules and other conditions described in the rules themselves. A program transformation as defined by the globalization equation preserves meaning only if those conditions implied by the substitution rules are met. Therefore these checks must accompany the program transformation to preserve meaning. Like the queries for the objects to be compensated, checks are non-destructive, so they can be mapped inexpensively through m and m^{-1}. These checks fall into the following categories.

Checks implied by the rules. For one, each rule depends on the dependence graph having particular properties. The most prevalent property is that the distributivity rule, the second case of the transitivity rule, and the indirection rule cannot be applied if there are multiple definitions of the variable (i.e., multiple walks labeled by the variable) reaching the vertices being modified.

Also, the application of a substitution rule must be consistent with the contour rules. For instance, as shown in the example under the extension rule, the new contour introduced by the distributivity rule must not violate extension. In this sense distributivity is not really a complete rule, nor is transitivity for the same reason.

Other implied checks are discussed below in the paragraph **Move checks**, below.

Checks because the general rule is not supported. Other preconditions are required because the transformation does not implement the most general compensation implied by the substitution rules. The restricted application of the supporting rules requires checking that the situation does not fall into the more general category. For example, to inline expressions `var-to-expr` copies the defining expression to be inlined. This copy-and-split paradigm is justified by the distributivity rule, which specifies that a split edge must have its variable label changed. Thus when `var-to-expr` copies a whole expression (rather than a single vertex), the rule should be applied for each vertex in the expression, but it is not. The result is that if the expression has a side-effect on a global variable, then evaluating both the original and the copy will not in general preserve meaning. This is what must be checked.

There are two reasons why a wholesale copying can succeed at all. (1) The copied expression passes values between expressions using nesting rather than a variable, so in the PDG there is no label on the edge that is passing the value, obviating the label change. (2) The variables used for passing values are declared locally to the expression, so the variable is implicitly changed by the copy.

A more sophisticated version of the copy operation would allocate a new variable for the side-effected store to make the split total. However, in some cases this is not practical (or desirable) because the expression contains a call to a procedure that side-effects a global variable. If this kind of split is desired, it must be done explicitly by the engineer, rather than as a natural part of the `var-to-expr` transformation.

Another situation requiring a check is when not every type of dependence is compensated. For most transformations, only one of data flow-dependences or control flow-dependences are compensated. For example, `var-to-expr` only compensates across the data flow-dependences generated by the variable uses of definition the expression being inlined. If there are other types of dependences that are affected, the transformation is terminated. Def-order dependences are a prime example, as they link (some) multiple definitions for a single use.[9]

Program syntax constraints. Many of the syntactic constraints are satisfied directly by the PDG and the contour rules, as described in Section 4.2. The rest are simply not implementable as program transformations. To use the example from Section 4.2, it is not legal for a program expression to return values to two places, and this constraint is implied by the fact that the parent of the expression gets the result. However, it is also not possible to construct a program that has any other property. Thus this kind of feasibility constraint is automatically satisfied during the lift to the program transformation.

The remaining class of syntactic check just assures that the transformation actually applies to the object. For example, `var-to-expr` only works when applied to a variable. This is declared in the interface of the transformation procedure, which requires an AST variable expression. Checking the syntactic legality of moving an expression is a slight exception. Many constructs require that some expression be present in a location. For example, the condition for an `if-then` cannot be omitted. Thus it is not syntactically legal just to take the conditional expression of an `if-then` and put it somewhere else. Only objects in sequence constructs such as scope declarations and scope bodies can be moved. Although this could be checked by the syntactic interface of the `move-expr` transformation, it is more easily checked dynamically.

Implied by the rules, part II: Move checks. As a special case of uncompensated edges, anti-dependence and output dependence edges only make it easy to check if moving code will add a new dependence. These edges are affected only by changes that reverse the order of the evaluation of the expressions linked by the dependence. This is the

[9]They link only some because def-order dependences apply only when one definition can actually overwrite the other, which means they cannot be in exclusive arms of a conditional [Horwitz et al. 88].

converse of data flow-dependence, for which the dependence is removed when reversing the order of evaluation of the linked vertices.

When a transformation moves an expression, it treats the violation of an anti- or output dependence as a hard constraint and terminates, rather then trying to compensate. Thus there is no substitution rule for a move—since it is not allowed to be a change to the PDG—except the implicit rule that any change not covered by a substitution rule cannot change the graph in any way. With the information provided with the failure of a move check (a labeled dependence between two expressions), it is possible to apply `expr-to-var` to the offending variable use (e.g, for an offending anti-dependence) to store it in a temporary that can be moved over the offending definition.

Control dependence is another special case that also comes in to play when moving an expression. A control dependence is sensitive to the syntactic extent of the controlling expression, rather than to just evaluation order. That is, an expression can get moved out of (or into) the arm of a conditional, changing the expression's behavior. In this sense control-dependences operate more like scope contours than flow-dependences. Another special quality is that it is an incoming (rather than outgoing) dependence that must be checked. (Note that although incoming flow-dependences can be broken by moving an expression, anti-dependences eliminate the need to look back). Because of these two qualities, control-dependences are checked like contours rather than dependence edges: the transformation checks to see whether the destination of the move is in the same control region as its source.

Another unique quality for checking a move is that all code that can be executed within the moved expression must be examined for offending dependences. This is because the execution order of every instruction executed within the expression is being changed. Thus, before performing a move check, the body of every function that can be invoked in the procedure must be examined for variable references that are visible in the scope of the context of the move. For example, when performing `var-to-expr` on the second use of `nr2` in the matrix-multiply example (See Chapter 2.6), the moved expression `(matrix-rows m2)` transitively invokes expressions within `matrix-rows`, which could affect the values within `m2`, since it is an array structure. The check reveals that there are no side-effecting operations in `matrix-rows`, and ultimately the move check succeeds.

4.4.3 Example: The Transformation for Inlining an Expression.

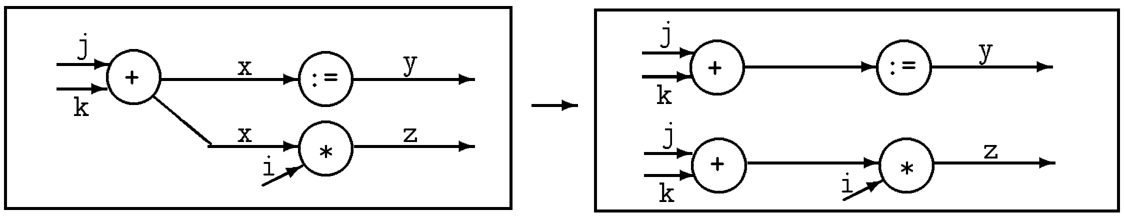
At the end of Section 4.4.1 the following manipulations for `var-to-expr` were proposed:

$$\delta p = \left(\prod_{u \in uses((\mathtt{v},e))} substitute_{(e,u)} \right) \circ remove_{(\mathtt{v},e)}$$

What justification can be provided that this preserves meaning? It is necessary to show that the above changes, with the appropriate checks, map to the above substitution rules. Using d_v to denote the variable definition being removed, e the expression defining v, and u_{vi} for the uses of the definition, the substitution rules for δg are:

- *Distributivity*, to allow replicating e and moving the edges of the original to its copies when there are multiple uses of d_v. This is a simple copy, so the check for e being side-effect free applies here.
- *Substitution*, to allow removing the label on the dependence between the inlined expression and its use. This is trivially applicable, since unlabeled edges are not constrained by contours.
- *Control*, to allow moving e to replace u_{vi} that are in a conditional.
- *Extension*, to allow moving e into scopes where any u_{vi} resides. All edges to and from the moved vertices must be checked for consistency.

Given the above substitution rules, the required checks are (1) the move checks for each inlining of the expression, including the extension check for scope dependence, (2) as implied by the distributivity rule, verifying that the inlined expression is evaluated



```
x := j + k
y := x
z := x * i
```

```
y := j + k
z := (j + k) * i
```

Figure 4.13: A flow diagram for the `var-to-expr` transformation

only once *or* that a simple copy produces an identical expression and (3) verifying that there are no extra variable definitions reaching the removed variable uses:

$$\left(\bigwedge_{u \in uses(v)} defs(u) = \{v\} \right) \wedge (uses(v) = \{u\} \vee identity(e, copy(e))).$$

Note that when incorporating the move check that it is possible to use the proof for `move-expr` as a lemma.

It remains to be shown that the above is sufficient to demonstrate that `var-to-expr` preserves meaning. It is necessary to show that the four groups of edges discussed above have been verified to be unaffected. (1) All data flow-dependence successors are compensated, and all other types of edges are verified to be unaffected. (2) None of the inputs to the inlined definition are changed, since only its final assignment is eliminated and the move check shows that none of its inputs are affected. (3) The compensation operations do not affect the output edges of the compensated objects. In particular, only the method of input is changed: the values arriving and the operation performed on them are unchanged. (4) Finally, it is verified that in the compensated object that there is only one data flow-dependence for the particular variable use being modified, so no input edges to compensated objects are affected.

4.5 Updating the PDG

It is desirable to restructure programs interactively. However, computing a PDG from program text requires polynomial time in the length of the program (See Chapter 5.6). Fortunately, because the restructuring transformations are meaning-preserving, there is an opportunity to develop efficient incremental PDG updates. In particular, the substitution rules for deriving a transformation can be applied directly to the PDG to track the transformation's changes to the program. This is efficient because the substitutions have a small impact on all dependences, meaning few updates are required to preserve dependence—one of the two qualities required to preserve meaning (the other being the preservation of the operations).

The rules describe the updates to data and control flow-dependences, but not anti-dependences and def-order dependences, which are affected indirectly by the updates. (The contour rules prohibit any change to the PDG, so they are not considered here.) Because direct flow-dependences are unchanged for an unmodified vertex, it is only necessary to consider the changes to anti-dependence and def-order dependence for vertices directly modified by the rule. Furthermore, any newly introduced variable (denoted `r` in the rules) cannot have an anti-dependence or def-order dependence associated with it, because there is only one definition of the new variable. This is what the updates look like:

Transitivity, case 1 $(ue_{\mathtt{s}_i}v_i \equiv ue_{\mathtt{r}}xe_{\mathtt{s}_i}v_i)$.

- *Anti-Dependence.* $e_{\mathtt{s}k} = FD(u, v_k)$ in G is $e_{\mathtt{s}k} = FD(x, v_k)$ in G'. This implies $AD(w, u)_{\mathtt{s}_k}$ in G must be $AD(w, x)_{\mathtt{s}_k}$ in G' since the definition (kill) of `s` has moved to x.

- *Def-Order Dependence.* Since control and data flow-dependence are unchanged for unmodified vertices, it is only necessary to consider vertices directly modified by the rule. Edge $e_{\mathtt{s}k} = FD(u, v_k)$ in G is $e_{\mathtt{s}k} = FD(x, v_k)$ in G', and x has the same control dependence as u, so $DO(u, w)_{\mathtt{s}}$ in G implies $DO(x, w)_{\mathtt{s}}$ in G'.

Transitivity, case 2 $(ue_{\mathtt{s}a}v_a \equiv ue_{\mathtt{s}a}xe_{\mathtt{r}}v_a, \ \nexists we'_{\mathtt{s}}v_a)$.

- *Anti-Dependence.* Edge $e_{\mathtt{s}a} = FD(u, v_a)$ in G is $e_{\mathtt{s}a} = FD(u, x)$ in G'. Thus $AD(v_a, w)_{\mathtt{s}}$ becomes $AD(x, w)_{\mathtt{s}}$. Since u has the same definition of `s`, it is not affected for anti-dependence.

- *Def-Order Dependence.* By definition, for this case, there cannot be multiple definitions of `s` reaching v_a in G, hence there can be no def-order edge caused by v_a's use of `s`.

Distributivity $(x_k ue_{\mathtt{s}}v \equiv x_k u'e_{\mathtt{r}}v, \ \ u' \overset{op}{=} u \wedge \nexists we'_{\mathtt{s}}v)$.

- *Anti-Dependence.* Edge $e_{\mathtt{r}}$ between uv in G must be $e_{\mathtt{r}'}$ between $u'v$ in G' to avoid multiple definitions of `r`, since u remains in G'. Thus, any $AD(v, w)_{\mathtt{r}}$ in G is gone in G'. Since $FD(u, v)_{\mathtt{s}}$ in G implies $FD(u', v)_{\mathtt{s}}$ in G', any $AD(w, u)_{\mathtt{s}}$ in G must be $AD(w, u')_{\mathtt{s}}$ in G'.

- *Def-Order Dependence.* As with the second case of the transitivity rule, there can be no def-order edge to update because of the prohibition of multiple flow dependences $e_{\mathtt{s}}$ to v.

The edge updates for the indirection rule are analogous to applying both versions of the transitivity rule. Reasoning about the updates for the control rule is more involved, but the result is similar.

Control, case 1 $(p_0\{p_1, xu_i\} \equiv p_0p_1\{e_{\mathsf{true}}xu_i, p_1e_{\mathsf{false}}x'u_i\}$, x' data congruent to $x)$. Since x' is data congruent to x, it must have all the same incoming data flow-dependences. It also gets copies of x's output data flow-dependences.

- *Anti-Dependence.* The anti-dependences in and out of x' must be copied from x with the data flow-dependences. Also, suppose that x (and hence x') uses and defines a variable `s`. Then putting x' in G' introduces anti-dependences between for `s`.

- *Def-Order Dependence.* As with anti-dependence, the existing def-order dependences of x must be copied for x'. However, the definitions of the same variable in x and x' are not def-order because (contrary to the definition of def-order) they are executed mutually exclusively, so no edge has to be added. There is one additional case. If there is an output dependence $OD(w,x)_{\mathtt{s}}$ (abstractly, since they are not in the implementation), it appears that moving x into the conditional will make it a def-order dependence. However, because x' is moved into the opposite arm of the conditional, there can be no reaching definition of $\mathtt{s}$ by w. So by definition there is no def-order dependence.

Control, case 2 $(p_0\{x, p_1 e\}u_i \equiv p_0 p_1\{ex, e\}u_i)$. In this case there is no replication of x because all uses of its values are in the conditional.

- *Anti-Dependence.* Conditional execution has no impact on anti-dependence, because there need be just *one* execution path from use to definition for it to exist. Thus anti-dependence is unaffected for this rule.

- *Def-Order Dependence.* Conditional execution does have an impact on def-order dependence, but in fact there is no change for this rule. In fact, since no variable definitions or uses are being modified and meaning is supposed to be preserved, def-order dependences *must not* change [Horwitz et al. 88]. However, it is still useful to show how def-order dependence and the rule interact. Suppose there is $DO(w,x)_{\mathtt{s}}$ in G that is missing in G'. The dependence implies by definition that $FD(w,u_a)$ and $FD(x,u_a)$, and that one of these conditionally overwrites the other. If the dependence is missing in G', then either (1) one of the data flow-dependences is gone, or (2) the control dependence on w or x has changed so that neither overwrites the other, or (3) the dependence becomes $DO(x,w)_{\mathtt{s}}$. The first case, the change to data flow-dependence, is not described by the rule, since only one control dependence is modified. This contradicts the assumption that the rule was applied to yield G'. The second case implies that the two definitions must be in exclusive arms of a conditional in G'. Since this rule is moving x under p_1, this implies that w must be under the opposite arm of the same conditional p_1. However, this implies that w's data flow-dependences cannot reach u_a, since it is, by definition of the rule, under the other arm of the conditional. This contradicts the existence the edge $FD(w, u_a)$, so there could not have been such a def-order dependence in G. The third case implies that the control dependences of *both* w and x have been changed, so that the flow-dependences have survived in G', as has the conditional overwrite. However, this contradicts the assumption that the rule was applied to yield G', since w is not modified by the rule. The reasoning works similarly for a def-order dependence in G' that is not in G.

Control, case 3 $(p_0\{xe_{\mathsf{s}}, p_1{}^*e_{\mathsf{true}}\}u_i \equiv p_0p_1{}^*\{e_{\mathsf{true}}xe_{\mathsf{s}}, e_{\mathsf{true}}\}u_i$, such that $\nexists\ we'_{\mathrm{LC}(p_1):\mathsf{s}}u_a \vee xe_{\mathrm{AD}:s}x \ \vee\ p_1{}^*p_i{}^*ve_{\mathrm{AD}:var}x)$. Like the case above, there is no replication of x, but x is moved under a circular control dependence, meaning that expressions later in the loop may affect x. Such effects are prohibited by the conditions in the rule. Thus it is necessary to consider only the iterated evaluation of x. In particular, the iteration of x results in multiple uses of its input flows, and multiple definitions of its output flows.

- *Anti-Dependence.* The case of x's affect on itself is prohibited by the rule. Likewise by anything else in the loop on x. However, now s can now be defined after the u_i as well as before. This means that for all data flow dependences xu_i, there must also be added $u_ie_{\mathrm{LC}(p_1)\mathrm{AD}:s}x$.

- *Def-Order Dependence.* As described for case 2 of the control rule, def-order dependence must not be affected for a change. The reasoning is similar in this case, so it is not repeated here. All those cases that could be affected by the addition of looping are prohibited by the conditions, such as disallowing variable definitions of s inside the loop.

This locality of updates puts an upper bound on the number of updates required for applying a substitution rule, assuring efficient update of the PDG for any transformation in the tool. For example, in case 1 of the transitivity rule, the updates are counted as follows: Create one new assignment vertex x, and one edge from u to x, a control dependence edge for x, and $|v_i|$ flow dependence successor edges from u are updated to originate from x. Also, the anti-dependences of u must be moved to x, whose number is bounded by the number of uses of `s`. The number of def-order dependences moved from u to x is bounded roughly by the maximum static nesting depth of conditionals in the program, times the branching breadth, times $|v_i|$, the uses reached by u.

4.6 Summary

An abstraction for handling the flow-dependence and scoping of a program, derived from a commutative diagram of the AST and PDG, has been defined. The abstraction has made it possible to locally describe a class of structural changes to a PDG, and hence, through the globalization equation, to its program. The changes are substitutions of subgraphs based on the equivalence of vertices and the modification of the scope structures that delimit flows. The globalization equation helps relate transformations in the program to substitutions in the PDG, avoiding program reconstitution by PDG unparse techniques. This assures that an untransformed form in the original program remains unchanged in appearance. Globalization also allows applying a program transformation and its corresponding PDG transformation together, avoiding expensive batch reconstruction of the PDG.

The focus of transformation on the AST (with the PDG used for reasoning) enables preserving meaning by predictably modifying structure, rather than violating it. In particular, the compensations are designed by the tool designer to be structurally and syntactically coherent. For example, restructurings do not change scope structure or variable names, unless they are explicitly designed to do so. Compensations are not dynamically sought by the system in an attempt to preserve meaning. The static choice of compensation transformations is not only efficient, but practical for good design of restructuring transformations.

Chapter 5

Implementation

The model presented in the previous chapter is the roadmap for implementing the abstractions of program representation, maintaining the relationships between them, and supporting the obligations for reliable meaning-preserving transformations. In fact, the implementation is a realization of the model's commutative diagram for program and PDG transformation. Also, the model's globalization equation (Chapter 4.4.1) guides lifting a PDG transformation δg to a program transformation δp, so that a transformation on program P is not reconstructed dynamically from PDG G via the m mappings. However, searches are still performed via the AST-PDG mapping functions m and m^{-1}. Thus the implementation must support representations for P and G, and for m and m^{-1} sufficient for doing search. These, respectively, will be an AST, PDG, and relations implemented in a tabular form to connect their data elements.

The model used globalization for lifting a PDG transformation to the AST, giving precise control over the effects of a PDG change on the program, and also making efficient transformation possible. Lifting PDG queries to the AST in a similar fashion has the advantage of allowing queries to be formulated on the rich mathematical basis of data flow analysis and PDGs, while providing the familiar notation of program syntax. It also frees the builder of a transformation from the details of how to perform mappings and the mechanics of the underlying representations. Abstracting away from the PDG is implemented by building a query layer on top of P, G, m and m^{-1}.

For the transformations themselves, applying the δp and δg functions in parallel on P and G respectively, as suggested by the commutative diagram, is sufficient to avoid the P–G mapping considerations, except that the representations of m and m^{-1} must be kept up-to-date. These parallel updates can efficiently implement transformations, avoiding the batch reconstruction of the AST or PDG that is normally used to keep these representations consistent [Horwitz et al. 89] [Larus 89]. Although incremental update has been implemented for only two transformations, `move-expr` and `rename-variable`, there is no inherent problem with applying the idea more broadly.

Auxiliary to the desire to ease implementing transformations and providing good performance was the wish to design the tool's components with enhanced functional independence. This would permit rapid prototyping of research ideas and using an existing PDG implementation to avoid unnecessary coding. One prototyping experiment, described in this chapter, was implementing incremental PDG update for the `move-expr` transformation. This helped validate the claim that efficient incremental update is possible.

This chapter describes an implementation of the abstractions in the model that meet the additional constraints of good performance and ease of change. The use of `var-to-expr` as an example is continued here, showing how it is implemented. The details of keeping the AST, the PDG, and the mappings between them consistent are discussed at the end of the chapter. Before getting into these details, the implementation platform and the system's overall structure are discussed.

5.1 System Minutiae

The prototype tool supports the restructuring of Revision 3 Scheme programs. It is implemented in Common Lisp (CL) and the Common Lisp Object System (CLOS). The PDG implementation is a subsystem of Curare [Larus 89]. It supports interprocedural analysis, including the aliasing properties of list structure references[Larus & Hilfinger

88]. The PDG supports all features of Scheme except `eval`, first-class functions, continuations, and dynamic scoping. With a more complete PDG implementation the tool could support these features (see Chapter 7.4). The tool also does not support macros and minor syntactic variants. The PDG module accounts for about 17,000 lines of the system, and the total system amounts to about 31,000 lines.

In the following, the terminology of CL and CLOS is used extensively. The tool's hierarchical type and implementation structure is described by a set of CLOS classes, and methods implement units of function. Roughly, a CLOS function-call dynamically selects a method to invoke by choosing the one whose formals best match the types of the actuals supplied. Also, a single call may invoke several methods (in some linearized order), one of which is a primary method, and the rest auxiliary methods. This allows modifying the function of a tool by writing an auxiliary method rather than modifying existing methods.

For example, consider the following two method definitions of `myadd` with parameters `x` and `y`, one that takes two integers and one that takes two reals:

```
(defmethod myadd ((x integer) (y integer))
  (float (+ x y)))

(defmethod myadd ((x float) (y float))
  (+ x y))

(myadd 1 2)
(myadd 1.0 2.0)
```

Which is selected depends upon the types of the values supplied. The first call of `myadd` invokes the first method; the second call invokes the second method. An auxiliary `:around` method can be added to these with this definition:

```
(defmethod myadd :around ((x integer) (y real))
  (call-next-method (float x) y))

(myadd 1 2.0)
```

With this added declaration, the call following it will go to the around method before invoking any other `myadd` method (the ones above would not match, anyway). The `call-next-method` then calls the method matching the types of the parameters, which now would be the version taking two floats.

In the context of the prototype implementation, a module is a logical grouping of classes and methods, typically stored in files separate from the code of other modules. Except for the PDG implementation, which is actually implemented as an independent CL package, no explicit import-export structure is used to gain access to interfaces.

5.2 System Overview

To put the following sections in context, consider Figure 5.1, a picture of the intermodule dataflow structure of the prototype, with annotations (in italics) relating the modules back to the model.

The low levels of the system represent the fundamental model abstractions. The AST module is the P, the PDG is G, and the AST-PDG mapping and consistency modules are, basically, the m and m^{-1}. There are two parts to the m function, the constructive component, which parses AST forms into PDG representation, and the data maps from existing AST nodes into PDG vertices. The data maps are constructed by the parser. The AST-PDG consistency module is more of an implementation detail, being responsible for keeping the AST and PDG consistent when the AST is changed by a transformation. Consistency is implemented with an event-mediator integration mechanism [Sullivan & Notkin 90] that allowed decoupling the implementations of the AST and PDG, but supports powerful integration of their respective functions. Currently consistency is reestablished by reconstructing most of the PDG from scratch, although the mechanism has allowed migrating to incremental updates in two cases.

The Flow query and Scope contour modules, combined with the AST, form the program representation layer. PDG operations are available through a syntactic interface, simplifying access to semantic information. This layer supports the Restructure module, which consists of layers of syntactic transformations, syntactic and semantic checks, composed using globalization to create meaning-preserving transformations.

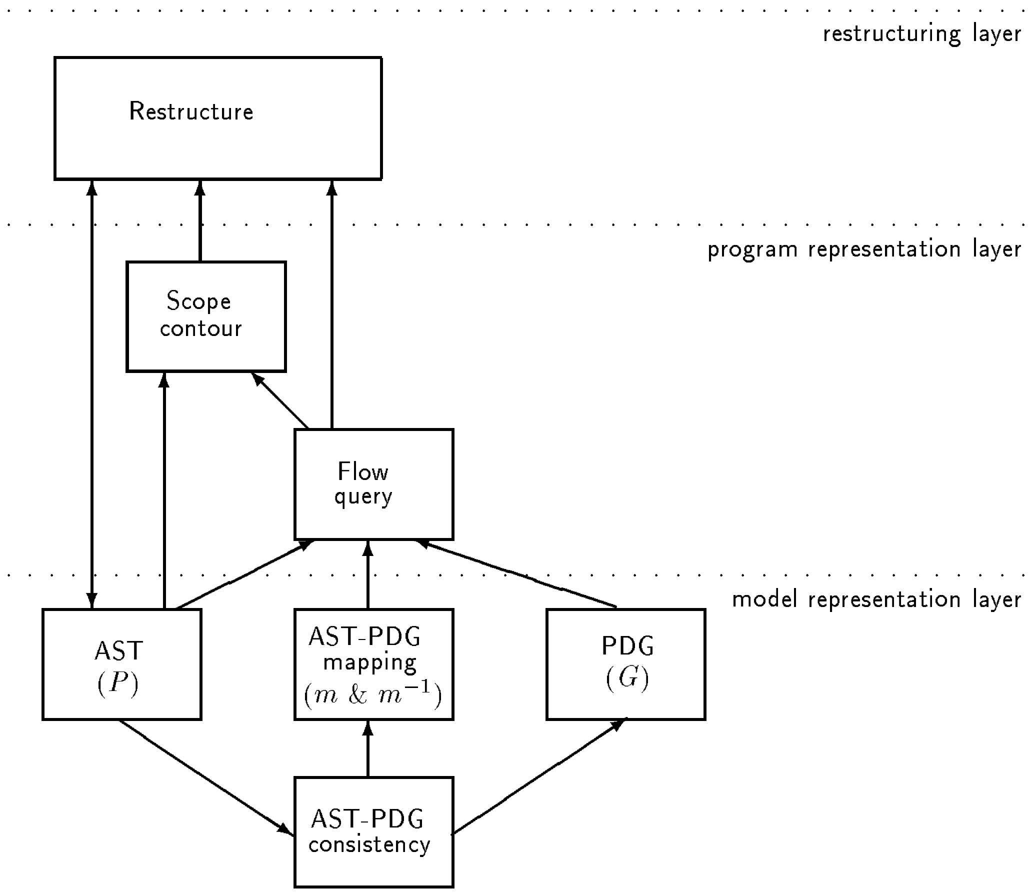


Figure 5.1: The prototype's inter-module dataflow diagram

A meaning-preserving transformation has the general structure of (1) check to see if transformation can succeed, (2) transform if legal, (3) report failure if not. The check is a set of primarily semantic queries accessing Scope contour and Flow query that assures that the transformations on the objects will be sufficient to preserve meaning. The transformation is typically a sequence of local changes to the AST, although it often contains queries to Flow query to obtain the objects that must be updated to preserve meaning. Also, each top-level transformation must assure that the AST and PDG are up-to-date. A transformation handles this by warning AST-PDG consistency before transformation that they must be consistent, and then after transformation telling AST-PDG consistency what changes it has made and that it is done transforming.

5.3 The Model Abstractions

P: **The AST.** The core data structure of the prototype is the abstract syntax tree (AST) module (See Figure 5.1). Its design is similar to the one described in Chapter 4.2. The AST is a widely used program representation in programming environments. It closely matches a program's textual representation, providing a convenient representation for managing syntactic relationships and the semantics of scoping [Aho et al. 86], as well as modifying the source of the program. It is created from a simple top-down parse of the Scheme program's s-expressions. Each node in the AST is a program object, such as a variable, function definition, or expression. A node has children if it is a composite object. For example, a function call's children are the function to be called and the expressions that make up the arguments.

G: **The PDG.** The data of the PDG module is derived from the AST. The PDG, as described in Chapter 4.1, is used to reason about data and control flow, and is derived from low-level data-flow and control-flow analysis. The first step in constructing the PDG is to translate the AST with a parser into sequences of statements that are essentially compiler triples [Aho et al. 86]: $(operation, result, arguments)$, where $operation$ is either operator call, function call, a jump or a jump label, $result$ is a variable to hold the result, and $arguments$ are the variables that contain the inputs to the operation. From this is computed a graph of the program's control flow, called the Control Flow Graph (CFG) [Aho et al. 86][Ferrante et al. 87][Larus 89]. Abstractly, computing the CFG removes all the jumps and labels and replaces them with explicit edges linking basic-blocks of these triples. Finally, building the PDG involves computing all the data relationships and necessary control relationships for each triple, and then storing these relationships in a PDG node along with the triple.

Mappings m and m^{-1}. These two mappings between the AST and PDG enable searches for AST objects using the PDG, as needed in the globalization equation. To support these, the AST-PDG mapping module contains two invertible relations between AST objects and PDG objects. The primary one is a mapping between AST variable and literal references and CFG variable and literal references in a triple (See Figure 5.2). A variable reference is either the definition or use of a variable in a triple underlying a PDG node (including the variables representing operations), and so is a natural object to map. This is a binary relation (*ast-object*,*pdg-object*) such that given an *ast-object*, the *pdg-object* is retrieved, and *vice versa.*

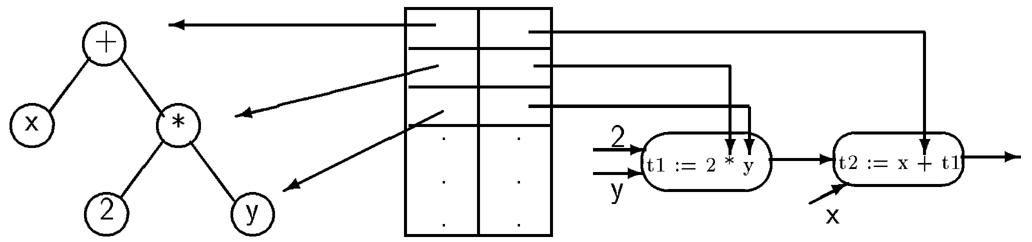


Figure 5.2: Part of variable relation between the AST (left) and PDG for `x + 2 * y`

Because some AST forms are normalized during translation to the PDG, a single AST variable reference can denote two or more PDG variable references. For this reason, actually a *list* of PDG variables is associated with an AST variable.

One change was required to the PDG implementation to resolve the dichotomy between mapping variables and searching based on PDG nodes. When searching the PDG based on some AST expression, it must be mapped to a PDG node (or set of PDG nodes) for the search. But in some cases, once an expression is mapped to a PDG node, it can be difficult to separate search initiated from a node's operation or its arguments. The standard problem is mapping from a simple use of a variable in the AST. This is not represented as an operation in a CFG triple; it is folded into a triple including variable references (uses) to be used arguments to an operation, a program operation such as `+`, and a store of the result into a variable. To separate the search based on a variable use in the AST, each actual program variable (and literal) must be assigned to a temporary before being used in an operation, thus giving each AST expression its own triple. Likewise for assigning the result—an operation always assigns to a temporary first, and then to the real program result variable. This results in each AST expression, even a variable use, being mapped to its own PDG node. Literals and operation result variables were already handled in this way in the original implementation of the PDG, but variables were not assigned to temporaries. The change was simple, and basically involved not treating literals and variables differently during parsing.[1]

This problem is exhibited in Figure 5.2, which does not use temporaries variables between nodes to decouple expression mappings. If a search were initiated on the reference to `y`, it would be mapped to the `*` operation. A search would confuse the call of `*` and the reference to `y`. The PDG in Figure 5.3 corrects this mismapping by giving `y` its own PDG node.

The second relation is between AST expressions and PDG (temporary) variable references. This relation is used to find the flows due to the value returned by an expression, which are represented in the PDG by temporary variables (abstractly these represent unlabeled flows). Also, if a search in the PDG yields a temporary, this relation maps it back to an AST expression. The reason that the two relations cannot be put into a single relation is that treating an AST variable like an expression could yield a PDG temporary it was assigned to, when in fact the actual PDG reference of the AST variable is desired. The relations are packaged as three functions: `find-pdg-vars`, which maps to the PDG variable references associated with an AST variable reference; `find-pdg-expr-vars`,

[1]If the tool were reimplemented from scratch, this might not be the design choice made; the mappings could be reimplemented to map to edges as well as operations.

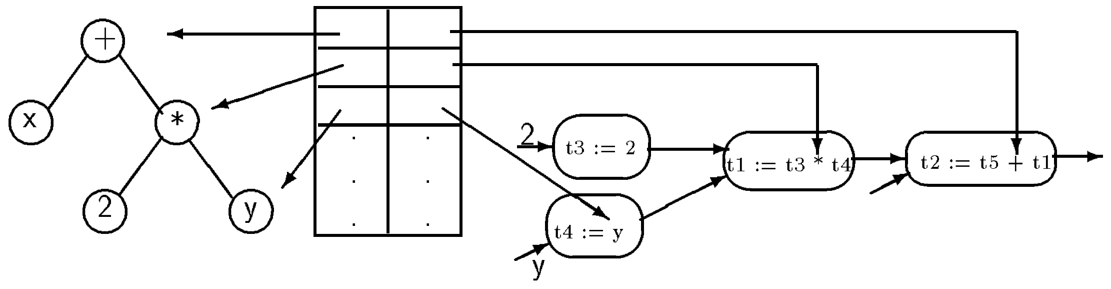


Figure 5.3: Part of variable relation between the AST and factored PDG for `x + 2 * y`

which maps to the PDG temporary variables for storing the result of the corresponding AST expression; and `find-ast-expr`, which maps from a PDG variable reference to either (1) an AST expression, if the PDG variable is a temporary variable denoting the value of the AST expression, or (2) an AST variable, if the PDG variable denotes an actual program variable.

Of course, variables are not the only objects in the PDG that need to be accessed. Abstractly, m should be able to map any AST object to any PDG object. Based on the context of the operation that needs the mapped data, the mapped variable can be translated into the object actually needed. For example, to access the outward graph edges due to a variable definition, parent pointers are followed from the PDG variable up to the PDG node representing the operation causing the definition. Then the labels of the successor edges can be matched against the variable to select the appropriate edges. For example, consider finding the PDG definition of a function based on its lambda definition. Mapping the AST lambda expression to the PDG will yield a variable definition, perhaps a temporary. By calling `contained-in` the enclosing triple can be reached, which contains the PDG lambda definition in the argument position.

The change to the PDG was eased by the fact that relations are represented independently of the AST and PDG. It has also allowed changing and adding relations without modifying the existing modules. Originally just the variable relation was implemented.

Later the expression relation was added, and soon after it was changed from a 1-1 relation to a 1-many relation.

One potential disadvantage of external relations like these is that when an AST object is removed from the AST, its entries must be removed from the relations to avoid garbage accumulation.[2] In this instance it is not serious, since it is easy to visit all variables in the AST or PDG to determine what is no longer reachable.

The relations can be implemented easily (and efficiently), by a pair of hash tables, one keyed on *ast-object*, the other on *pdg-variable*. The cost in space and time for hashing structures is minimal, although it is dependent on the exact implementation of the tables. Self-organizing hash tables [Larson 88], for example, have amortized cost of space linear in the number of elements, and constant time for lookups, insertions and deletions.

The relations are constructed by the AST-CFG parser (shown in the diagram as part of AST-PDG consistency). This parser updates the control flow graph (CFG), the base data structure of the PDG, and so it is the one module that knows how to build (and thus map) between AST and PDG objects. The details of building this mapping and keeping it consistent is described below in Section 5.6.

5.4 Mapping Operations on G to P

Because the AST is the central representation upon which contours, the PDG, and program transformation depend, but semantic information is central to restructuring, it is desirable to have a complete, AST-oriented interface for performing all queries and transformations. This, in essence, combines the best of the two sides of the commutative diagram, representing them through a single interface by completing the lift of G's operations to P. The mappings between the AST and PDG, make this possible. In particular, an implementation of a semantic operation on syntactic data only needs to

[2]The prototype's implementation of relations uses CLOS's features to dynamically add slots to the AST and PDG node types. This maintains separation without requiring a free-standing relation.

map the syntactic form into its semantic counterpart, perform the query, and then map back. This is easier than lifting transformations because the maps already exist as concrete element-wise relations.

The translation of semantic queries is the primary purpose of the Flow query module, but a secondary role is allowing queries of the form "What if...?" This is not simple because the query almost necessarily asks questions about the effects of introducing a construct that does not yet exist. The solution is to generalize the program representation interface by not requiring the existence of the AST form being queried about, but only its parts.

Several of these unified, generalized functions that comprise the unified program representation interface are listed below in Table 5.1. A handful of these are described and used throughout the chapter, giving a sense of the uniformity of the interface across syntactic and semantic queries. The right-hand column in the table says whether the query actually accesses the PDG. The query `crash-free?`, for example, does not. This query is useful if moving an expression will cause its control dependence to be loosened—it will be executed more often. If the expression cannot crash the program, then the move preserves meaning. Otherwise, loosening the control dependence risks causing an error that would not have occurred in the original version, because the expression now executes possibly more frequently, and may crash in one of these newly added executions. The reason that this query does not access the PDG, however, is only because it is very conservative: it checks that the expression is a constant, which is a syntactic property. A more ambitious check might actually check if the operations involved have crash-free properties, requiring an access to the PDG. With the interface as it is defined, this change in implementation would be transparent to the higher layers, making the change easier.

Translating queries on G to be on P. According to the model translating a semantic (PDG) query Q to a syntactic form should look like $m^{-1} \circ Q \circ m$. Because of the mapping relations, this is essentially the structure of unified queries that access PDG information.

The only catch, as mentioned in the previous section, is that the maps are formulated on variables, so other objects have to be retrieved with some additional work. Fortunately, mapping these operations to the AST also hides this wart. The advantage to structuring queries this way is that Q can be formulated on the rich mathematical basis of data flow analysis and PDGs, but it can be accessed in the familiar notation of program syntax.

To give a sense of how the mapping works for a query, consider `get-uses`, the syntactic translation of the `uses` query, first considered in Chapter 4.4.1. As defined there it was described as the data-flow successors to a PDG vertex due to a particular variable use. Its translated version in the model is $m^{-1} \circ \mathit{flow\text{-}successors} \circ m$, and it was used to globalize `var-to-expr` by finding the syntactic uses that depend upon the variable definition being removed. To implement this in the prototype, `get-uses` actually uses the def-use annotations derived from the CFG. The dependence edges of the PDG could be used here, but that would be unnecessarily complicated, given the presence of the def-use graph. `get-uses` retrieves the PDG variable definitions representing the AST variable, reads the `uses` attribute of the definitions, and then maps the uses back to the AST. This is shown in a simplified form in the following definition:

```
;;;
;;; Syntactic translation of the USES query.
;;;
(defmethod get-uses ((ast-var s-variable))
  (let* ((pdg-defs (find-pdg-vars ast-var))
         (pdg-uses (mapcan #'uses pdg-defs))  ; short, don't go to PDG
         (ast-uses (mapcar #'find-ast-expr pdg-uses)))
    ast-uses))
```

This is straightforward because the mapping and query are appropriately decomposed: the query is made on the representation best suited for it, and the data is formulated and delivered in terms of program syntax, which is familiar to the programmer. After performing this mapping `var-to-expr` only needs to apply the basic transformation on the definition, and the compensating transformations to the uses. This is shown in the next section.

Now consider the semantic query `strong-syntactic-dependences(expr1,expr2)`, which searches for a walk of data flow and control flow dependences that might link the two expressions semantically (See Chapter 4.1.2). It is one of several checks required before moving any expression, since a semantic link in the two expressions implies their evaluations cannot be reversed. Again, the PDG is the best representation for performing the search, since it explicitly contains the dependences, but the AST is the representation being manipulated. First, all the AST variable references in `expr1` are retrieved, and these are mapped to the PDG by the Flow query module using `find-pdg-vars`. This must include externally visible variable references from the body of any function called. Then parent pointers are followed from these variables using function `contained-in`, eventually reaching their containing PDG nodes. The nodes in `expr2` are gathered in a similar fashion. To walk from `expr1` to `expr2`, a transitive closure of data and control flow dependences, accessed by `successors`, is followed. If `expr2` is reached, then potentially there is a semantic dependence. In this case the walk of PDG nodes is converted back to its syntactic form: `underlying-object` is called to retrieve the triple, whose `var` slot is converted back to an AST expression or variable with `find-ast-expr`.

Generalizing the syntactic interface. By translating every semantic query like those above into its translated form, knowledge of the PDG's conventions are abstracted away, leaving a single, syntactically-oriented interface for queries. But now consider the second role of the Flow query module, allowing queries about program components that do not yet exist. This requires minimizing the dependence of a query's interface on the AST representation of the object being discussed. For instance, an argument to `strong-syntactic-dependences` can be merely an ordered list of expressions, but might become the body of a function, should the dependence check succeed. In the case where a structure is going to have a sub-object removed from it, the result of a query on the structure might require the impact of the sub-object removed using a set-differencing operation, or fashioning an alternative formulation of the query that bypasses the object.

For other checks the easiest implementation temporarily transforms the AST, performs the query, and then transforms back. An example is the `binding-dependences` check, which determines if moving an expression will change any of the bindings referenced in the expression. In the special case that a binding itself is being moved, it is also necessary to check that no other expressions are affected. The easiest way to check this is to capture all references to the binding in the old location, move the binding, then capture again all references to the binding, and move the binding back. If the difference of these two sets is non-null, the binding cannot be moved. If all the checks succeed, then the normal `move-expr` is called.

In Chapter 4.2, contours were introduced as an extension of PDGs to allow reasoning about scope structure when transforming the PDG. Since queries are initiated on AST objects, and the results reported via AST objects, many contour operations actually do not need to access the PDG. However, since queries are initiated from AST objects, this fact is transparent. One exception is the query `flows-out-of-scope`. This method takes a variable instance named v in the AST and a scope s, and the method determines if there is a definition of v in s that is used outside s. By mapping s into the PDG and traversing the flow dependences with labels v originating in s, and then converting back to the AST, it is then possible to see if the resulting AST nodes are in a scope other than s. In fact, the containment check could be performed in either the AST or PDG (by using the maps). The crucial point is that the scope structure is present only in the AST, and so the set of PDG nodes that is contained within s's contour is determined by the AST's scope structure. With the AST-PDG mapping this imposition of structure is trivial.

5.5 The Restructuring Layer: Realizing δp

The lowest levels of the system provide ways of relating the AST and the PDG and keeping them consistent, and the layers above use these to build a flexible syntactic interface for accessing all information. The remaining layer implements meaning-preserving transformations. The globalization equation of Chapter 4.4.1 prescribes that the pro-

Unified Syntactic Interface	Access PDG?
(built-in-fnct? ast-var)	Y
(user-fnct? ast-var)	Y
(var-use? ast-var)	Y
(var-def? ast-var)	Y
(get-definitions ast-var)	Y
(get-uses ast-expr)	Y
(get-values ast-var)	Y
(single-element-aggregate? ast-expr)	Y
(makes-list? ast-expr)	Y
(makes-vector? ast-expr)	Y
(crash-free? ast-expr)	N
(side-effects? ast-expr)	Y
(io-effect? ast-expr)	Y
(syntactically-movable? ast-expr)	N
(result-used? ast-expr)	Y
(control-parent ast-expr)	Y
(dominate? ast-expr ast-expr)	Y
(strong-syntactic-dependences ast-exprs ast-exprs)	Y
(in-scope? symbol ast-exprs)	N
(within-scope? symbol ast-exprs)	N
(get-binding symbol ast-exprs)	N
(defining-scope symbol symbol-location)	N
(defining-binding symbol symbol-location)	N
(all-flows-out-of-scope ast-exprs)	Y
(binding-dependences ast-expr new-location)	N

Table 5.1: Semantic functions with syntactic interfaces

gram transformations are derived by hand to be applied directly to P, and queries for finding elements to compensate is mapped through the m functions. The latter has already been abstracted away in the program representation layer. The globalization equation requires that the program transformation be justified by the substitution rules that parallel it in G and be accompanied by the checks implied by the equation and the substitution rules. This was discussed in Chapter 4.4.3, and the example below deriving `var-to-expr` is driven largely by these requirements.

A meaning-preserving transformation's top-level function is (1) check to see if can transform, (2) transform if yes, (3) report failure if not. Because consistency and translation are managed by the lower layers, a transformation's implementation typically adheres to this structure, and the restructuring layers (See Figures 5.4 and 5.6) have an analogous structure to this composition of operations. The structure of lower levels of function, such as globalization, was first discussed in Chapter 4.4.1.

The only deviation from this high-level view is the obligation to manage the propagation of changes from the AST to the PDG. Briefly (See Section 5.6 for details), a meaning-preserving transformation sends events at the beginning and end of a transformation to let the underlying AST and PDG know that they are about to accessed and should be consistent with each other. This allows rebuilding the PDG lazily. The events are sent automatically by a CL macro that wraps each transformation definition. Within the transformation is a method that performs the actual change to the AST. An auxiliary method is attached to this method that tells the consistency agent what has happened to the AST so that the PDG can be brought up-to-date.

High-level transactions and update signalling used with the program representation layer simplify globalizing transformations by keeping the details as far away as possible.

However, since a meaning-preserving transformation is the composition of a number of smaller transformations, it is possible that queries and transformations might be interleaved. Interleaved queries could access inconsistent PDG data, because logically the whole meaning-preserving transformation happens at once, with no intermediate state.

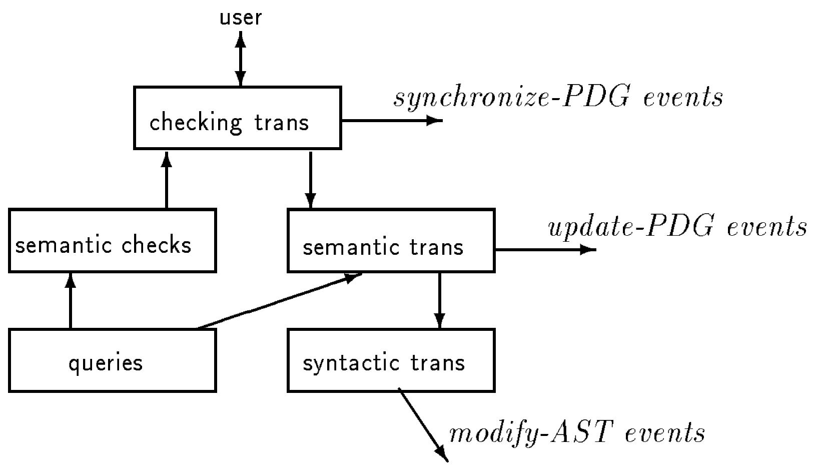


Figure 5.4: The layers within the Restructure module (Arrows represent dataflow)

There are two techniques that a programmer uses to build a transformation so that it does not access semantic information out of phase. By convention a transformation assures this within itself by retrieving all the globally related elements before performing any changes to the AST. But a compensating transformation called by it may try to access the PDG. However, it is possible to move the checks of the compensating transformations *before* any transformation begins. To allow this checks and transformations are implemented in separate methods. It is safe to have these checks up front because the main transformation and its compensations are logically done in parallel anyway, and hence the checks should really occur before either the AST or PDG have changed.

Because a transformation's check and the report of failure are separated (the check records the failure, but leaves other methods to give it to the user), it is possible for a top-level transformation to use the failure to guide alternative attempts to achieve the structure that the user desires. This is not exploited in the current prototype, primarily because failure-driven transformations have proven difficult to implement satisfactorily. For example, for `extract-function` to avoid failure, it might compute what variables will be out-of-scope when the function is moved to its new location, and force these to be parameters of the new function. However, this can yield a function whose interface is not the one desired by the engineer; the extra parameter might be for caching a temporary result (computed outside the extracted function) or for an embedded computation not part of the function the engineer intends to abstract.

This final caveat aside, implementing a transformation closely follows the construction introduced in Chapter 4.4.1, with additions for consistency of the AST and PDG data, most of which is automatic. Listed below are the tasks the transformation builder carries out, with comments about what was done to implement `var-to-expr`. The discussion has been simplified by assuming that the selected expression is a simple variable definition (not a variable use or a binding declaration) and eliminating some flexibility options.

1. *Implement the local transformation that the engineer logically applies.* Logically, the variable carrying the value of an expression is being deleted. Locally speaking this is the low-level operation `ast-remove!`.

2. *Implement the transformation that must compensate the local change.* According to the transitivity law, the compensation of deleting a variable definition is to insert the defining expression in all the places where the variable definition is used. The compensation is the low-level operation `ast-subst!`.

3. *Formulate the queries that select the forms to be compensated.* To retrieve the uses of the variable being deleted, `get-uses` is called on it.

4. *Write a transformation combining the local and compensating transformations.* This is the instantiated version of the globalization equation, called `var-to-expr` in this case, which takes the selected expression, finds its uses through `get-uses`, removes the expression with `ast-remove!`, and then does `ast-subst!` on the uses with a copy of the expression that originally defined the variable. This is a simplified version of the code:

```
(def-transformation var-to-expr ((def s-variable))
  (let* ((uses-of-def (get-uses def))
         (def-par (contained-in def))  ; an s-setting
         (def-expr (expr def-par)))

    (ast-remove! def-par (forms (contained-in def-par)))  ; local

    ;; compensating
    (dolist (var uses-of-def)
      (setq def-expr (copy-ast-node def-expr))
      (ast-subst! var def-expr))))
```

5. *Implement versions of this that signals modification events.* This is a wrapper that first stores away the scope expression that contains the definition of the variable to be removed, calls `var-to-expr`, and then calls `signal-dependents` on the scope. Below is the CLOS auxiliary method that transparently wraps the transformation.

```
(defmethod var-to-expr :around ((def s-variable))
  (let ((scope (defining-scope def)))
    (prog1
     (call-next-method)    ; call to the transformation
     (signal-dependents scope :CHANGE))))
```

6. *Gather the checks for the top-level transformation.* Basically, a check will examine the successors of the changing object, removing those that are being compensated, and those that are unaffected. If there is anything left, the transformation's compensations will not be sufficient to preserve meaning.

   The checks for `var-to-expr` are several-fold. First, the expression to be inlined must in fact be a variable definition, a `set!` for the simplified example. Second, moving the expression must not change its meaning. This requires calling `move-expr-check-semantics` on the defining expression with respect to each value returned from `get-uses`. This is called in all transformations that move code, saving considerable implementation effort for new transformations. Third, multiple evaluations cannot have side-effects. Thus if the result from `get-uses` has length greater than one, a false value is required from `side-effects?` on the expression. Last, none of the uses to be replaced can have two possible definitions (for example, one each in the arms of a conditional). This is checked with `no-multiple-defs-check`, which calls `get-definitions` (the converse of `get-uses`) on each variable use to make sure there is only one. The checks are put into `var-to-expr-check`. Below is a simplified version of the semantic checks.

```
(defmethod var-to-expr-check-semantics ((def s-variable))
  (let* ((uses-of-def (or uses (get-uses def)))
         (def-par (contained-in def))  ; an s-setting
         (def-expr (expr def-par))
         (movable-p (every #'(lambda (u)
                               (move-expr-check-semantics def-expr u))
                           uses-of-def))
         (copyable-p (or (not (side-effects? def-expr))
                         (<= (length uses-of-def) 1))))

    ;; Check to make sure that if it is a side-effecting expression that
    ;;  there will only be one of them.
    (when (not copyable-p)
         (report "Cannot multiple-substitute side-effecting expression: ~a"
                 def-expr))

    ;; Return a value indicating whether all the subchecks succeeded.
    (and (no-multiple-defs-check def uses-of-def) movable-p copyable-p)))
```

7. *Combine the top-level check and the transformation into a meaning-preserving transformation, using a macro that defines top-level functions with transactions.* This is the most simple step, and the code looks something like this:

```
(def-global-transform checking-var-to-expr ((var s-variable))
  (if (var-to-expr-check var)
      (var-to-expr var)
      (print-failures)))
```

The `def-global-transform` form puts a `(signal-sources)` call at the beginning that makes sure that the transactions are in phase and the PDG is ready to be accessed. At the end is inserted `(signal-restructure-complete)` to mark the end of the transaction. Note also that the check is separated from the report of failure, so that corrective measures can be taken in more sophisticated transformations.

As suggested by the implementation of `var-to-expr`, the lower level tasks often do not require implementing new operations, since they already exist. This is especially true for the syntactic transformations, and the basic semantic queries and checks. Also, since the basic mechanisms of consistency are in place, the declarations of AST changes and Restructure's order of access of the AST and PDG are usually simple. The major difficulties are in precisely stating checks, which requires knowledge of the semantics of the language and the semantic relationship between the old and new structure.

The next section provides the low-level details of implementing consistency of the low-level representations, which is followed by a brief overview of how all the system components fit together.

5.6 Maintaining Consistency Between the AST and PDG

As the AST is changed by transformations the PDG and the maps between them must stay consistent. Although incremental update of the PDG is possible, it has not been implemented across the board, requiring batch reconstruction of the PDG in the general case. This section describes the consistency mechanism and an experiment in implementing incremental PDG updates.

There are several conflicting constraints that together imply that consistency should not be embedded in any existing module:

- only the AST knows the effects of its operations on its own content,
- only Restructure knows when it needs semantic (PDG) *versus* syntactic (AST) information,
- only AST-CFG parser knows how to translate AST changes into PDG changes, and
- rebuilding the PDG is expensive, and so must be minimized.

Thus the AST contributes change information, Restructure knows about the ordering of AST and PDG access, the parser contributes knowledge about translating changes, and the PDG knows how to rebuild the PDG from the base CFG structure. Observe the intermodule dataflows in Figure 5.1 to visualize the interactions.

The consistency interactions are implemented in a special consistency module, AST-PDG consistency, using a relation of top-level forms between the AST and PDG and an early version of the event-mediator integration mechanism [Sullivan & Notkin 90]. Intuitively, an event is a signal from object a to object b that something has happened in a that might be of interest to b. The content of the event—its name and any associated data—provides the necessary information for b to act appropriately. By a sending an event rather than directly making a call on b, a does not require any prior knowledge of b. This can be implemented by keeping a list objects (b) that are interested in receiving another object's events (a). Also, so that b and a can be independently developed and used, a should not signal directly to b, but rather to a object that knows enough about a and b to do the updates.

This is the mechanism used to relate the AST and the PDG. Abstractly, the AST exports a number of events that it may signal, and describes the data that it will associate with each event—a type signature. In this implementation the events are not explicitly exported nor do they have an explicit signature. Acting as a mediator between the AST and PDG, AST-PDG consistency receives these events with the data described in the signature, and subsequently updates the PDG when semantic data is required by Restructure. Using events to signal change allows adding or removing modules dependent on the AST without modifying the AST. AST-PDG consistency becomes eligible to receive events from the AST by calling a registration routine.

AST announces three events: Insert, Delete, and Change. Each is associated with a single argument, the object affected by the operation issuing the event. In this implementation, the argument is always a Scheme top-level form. An insertion or deletion of a sub-form is converted into a Change of its containing top-level form. Although a

Delete followed by an Insert of a changed form are syntactically the same as Change, they are not semantically identical. The intermediate state between the Delete and the Insert might be inconsistent for a dependent tool. The event and registration structures were incorporated into the AST by adding to its superclass hierarchy, although relations could be used to associate registrants with AST objects. Signalling of events is handled by CLOS auxiliary methods, which allowed adding function to AST calls without modifying existing code. In a more traditional programming language it is necessary to have the event signalled at the end of the code implementing the mutating operation. The prototype attaches an auxiliary method to any method that mutates the AST, and the auxiliary method sends an event appropriate to the kind of change.

The PDG does not need to announce any events because all changes to it come through the event mechanism (See Figure 5.1). Since it does not announce events it was not necessary to modify the PDG to integrate it with the AST. This was valuable since the PDG was not implemented by the author.

To map an AST event to a PDG change, AST-PDG consistency maintains a relation between the top-level forms in the AST and CFG (See Figure 5.5; numbers match text). For a (1) Change event, it takes the AST data provided in the event, (2) maps it to the corresponding CFG data, and (3) replaces it with a reparse of the AST data. After such an update the PDG is no longer internally consistent. The Restructure module is integrated with the AST-PDG consistency to control when the PDG is brought up-to-date with the CFG (this relation is not shown in the system diagram). The following describes how this is implemented.

Lazy consistency and transactions. First, to assure that no *visible* state of the PDG is brought up-to-date with an AST in an intermediate state, AST changes are not propagated to the PDG until a transformation is completed; a sequence of changes by a method in the Restructure module is treated like a single transaction. Also, the cost of maintaining the PDG with batch construction is perhaps $O(|program|^3)$; the algorithmic complexity is not known, although it appears to be a function of the alias

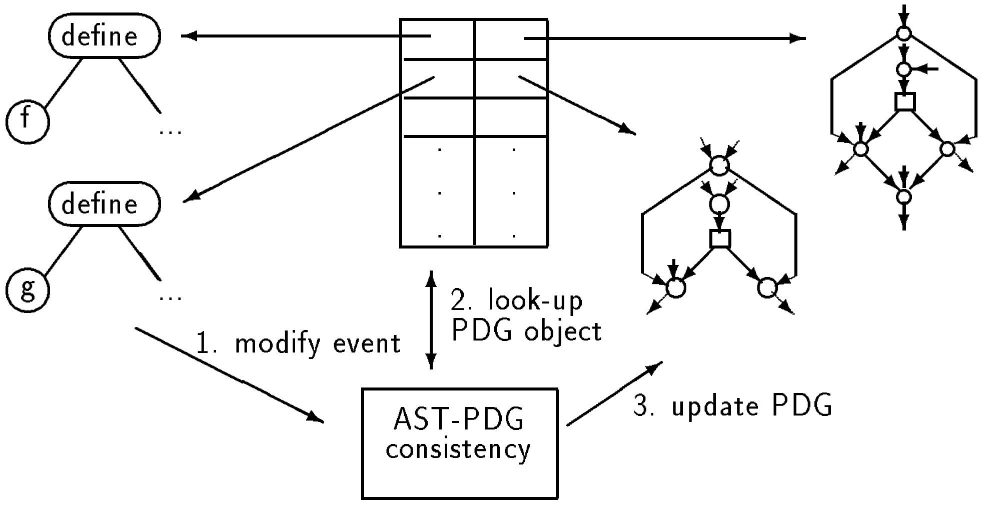


Figure 5.5: AST-PDG consistency translates AST events into actions on the PDG

data flow calculations.[3] Hence, the PDG should be updated only as needed and only where it is actually changed. (See Chapters 4.5, 7.4 and below for ways this cost can be reduced.)

Both concerns of consistency and performance require buffering the propagation of changes from the AST to the PDG. For AST-PDG consistency to buffer events correctly, it normally receives data from the AST, but does not do the update. When it receives a Transaction Complete event from Restructure, it is allowed to take the buffered messages and reconstruct the PDG from them. However, it does not do the propagate until it receives a Transaction Begin signal. This event means that the PDG is about to be accessed and the AST is about to be changed. At this point any cached events are propagated and the PDG is synchronized with the AST. If a Begin event is received within another transaction, but there have been events signalled by the AST in this outer transaction, this is an error, since presumably the AST is not in an internally consistent state. If a nested transaction—due perhaps to the transformation calling another

[3] J. Larus, Personal Communication, 1990.

transformation inside it—needs to be begun after some changes, and the tool builder has determined that the AST will be internally consistent, then an intratransaction message Synchronize can be signalled by the restructuring operation controlling the outer transaction. By not rebuilding the PDG until a Begin message is received, operations not involving transactions can take place without forcing a rebuild of the PDG. Also, under an incremental consistency scheme (see Chapter 7.4), transformations involving independent parts of the program need not force rebuilding unrelated parts of the PDG. Currently this is just a function of whether the transformation accesses the PDG—some local transformations and some transformations that focus on scopes do not.

This consistency mechanism lacks an undo facility to rollback the side-effects of an aborted transaction. In practice, this has required any transformation that is the sequential composition of other meaning-preserving transformations to move all the checks before any of the transformations. Moving the checks is potentially difficult because the checks for a later transformation usually assume the existence of a structure that does not yet exist. An advantage of implementing a composite transformation in this fashion is that it is not always necessary to rebuild the PDG between transformations.

Because of these impediments composite transformations have been implemented only when they have a sufficiently greater value as a composite transformation than as separate units. The best example of a composite transformation is `extract-function`. It is the sequence of operations `group-into-lambda`, which wraps an argumentless lambda literal around the code to be extracted; `abstract-bindings`, which gives parameters to the lambda and arguments to the call; and `expr-to-binding`, which moves out the lambda and assigns it to a variable so that it can be reused. The composition is useful because the sub-transformations are so frequently applied together that using it saves the engineer from unnecessary interactions with the tool. Also, the PDG does not have to be rebuilt between the subtransformations.

Although it is possible to move the checks forward for `extract-function`, it is complicated by the fact that the check for `abstract-bindings` does not have a lambda expression to query. The solution, described in general in Section 5.4, was to design the check so that it does not require a lambda expression, just a list of expressions that are (to be) the body of the function—the lambda is implied and so need not exist.

Temporary change is another situation in which lazy consistency is useful. Section 5.4 described the need to temporarily change the AST to simplify performing the `binding-dependences` check, among others. For such a change modify events need not be sent, which allows doing the temporary change at minimal execution cost. To support this requires that there be two layers of syntactic transformation—one that does not signal changes, and a layer built on top that does. For example, moving the binding for `binding-dependences` is handled by `syntactic-move-expr`, which is just like `move-expr`, but does not signal changes of the AST. This layering technique works because the move does not disturb the AST-PDG mappings and (due to single-threaded execution) the temporary state is seen only by the check.

Incremental consistency. As described above, consistency is maintained by reconstructing the PDG after changes to the AST that could affect the PDG's structure. But in Chapter 4.5 it was shown how the PDG can be kept consistent with the AST without batch reconstruction. This has been implemented for the `move-expr` transformation, revealing the process of making a transformation incremental and the problems involved. It also demonstrated the benefits of the system structure in allowing this quick change, which required an afternoon of work.

A move is syntactically simple in both the AST and PDG. To update the AST, all that is needed is a call on `ast-remove!` followed by a call to `ast-insert!`. A meaning-preserving move in the PDG should not change the PDG at all: no data flow or control flow dependences should be changed. However, the underlying CFG is affected, and since it is occasionally accessed on queries, it must be updated. The update requires finding all the statements (triples) corresponding to the moved AST expression, and moving the implied basic-blocks to new locations in the CFG. The complication is that some moving statements may be in a basic block that has statements *not* being moved. Splitting the offending basic block into two pieces before movement solves the problem.

To add this code to the tool only required replacing the typical `signal-dependents` call in the `move-expr` auxiliary wrapper method with calls to functions that perform the splits and moves described above. The change was isolated from the transformation and required no work-around of the consistency mechanism. No change was required to any AST or PDG code. However, this experiment did reveal that although the update is theoretically null, this particular implementation of the data flow information still required changes. It is likely that incremental updates for other transformations will have similar problems.

In fact, this PDG implementation is not designed for incremental update, and perhaps anticipating this change might have made it easier. The original consistency mechanism for the PDG is data-driven, but still batch oriented. The mechanism knows that when a certain class of information is needed, such as alias information, exactly what data flow analyses are required. It can then perform, for the whole program, just those analyses that have not already been performed from a consistent layer below. Similarly for recording changes, it was conceived that a transformation declare what representations it has changed, which invalidates the higher layers of representation. For example, a change to the CFG invalidates the def-use information and alias information, and above that the PDG—almost everything. There is no notion of how a small change in the CFG corresponds in the PDG.

Overall, the change was successful. Moves in the tool now require seconds, whereas before they required minutes. Since this is one of the most common transformations, it has improved performance overall. The experience with underlying representations suggests that queries should be implemented so that they do not access information below the PDG unnecessarily. Query `get-uses`, described in the next section, accesses the def-use information in the CFG. Avoiding these accesses would allow directly updating the PDG without concern for the CFG or other supporting information. This might require augmenting the PDG information to make some queries easier.

5.7 System Review

Now that the implementation and use of all levels of the system have been described in some detail, it is possible to describe how the upper-level of the system interacts with the lower levels. The modular structure of the system with dataflow arrows connecting them appears in Figure 5.1. The modules at the bottom—AST, PDG, and AST-PDG mapping—manage basic representation. The modules AST and Flow query define the high-level composite representation layer. The top-most module, Restructure, manipulates the program representation and controls consistency.

It is also instructive to see how the different layers and modules of the system are accessed during the execution of a transformation. Again `var-to-expr` is used as the example (See also Figure 5.6). (1) Transformation `var-to-expr` begins in Restructure by initiating a transaction, forcing AST-PDG consistency to read all the events that it has cached and translate them into updates of the PDG. (2) Then its check is invoked, which makes calls into the query and check layers like `strong-syntactic-dependences` described above and `no-multiple-defs-check`, the check required for many of the substitution rules (See Chapter 4.4.2). If the check succeeds, then the actual transformation is called. (3) Before any change is made to the AST, the uses of the variable to be replaced are retrieved by calling `get-uses` on the variable definition. Once these are safely in hand, the individual transformations from the local-syntactic transformation layer `ast-remove!` and `ast-subst!` can be executed. (4) At the completion of the transformation, the auxiliary method sends a Change event on the scope containing the binding of the manipulated variable. (5) When the top-level transformation completes, the transaction is signalled as complete.

5.8 Summary

The commutative diagram of the model, in conjunction with the globalization equation and the substitution rules, provided significant guidance in the implementation of

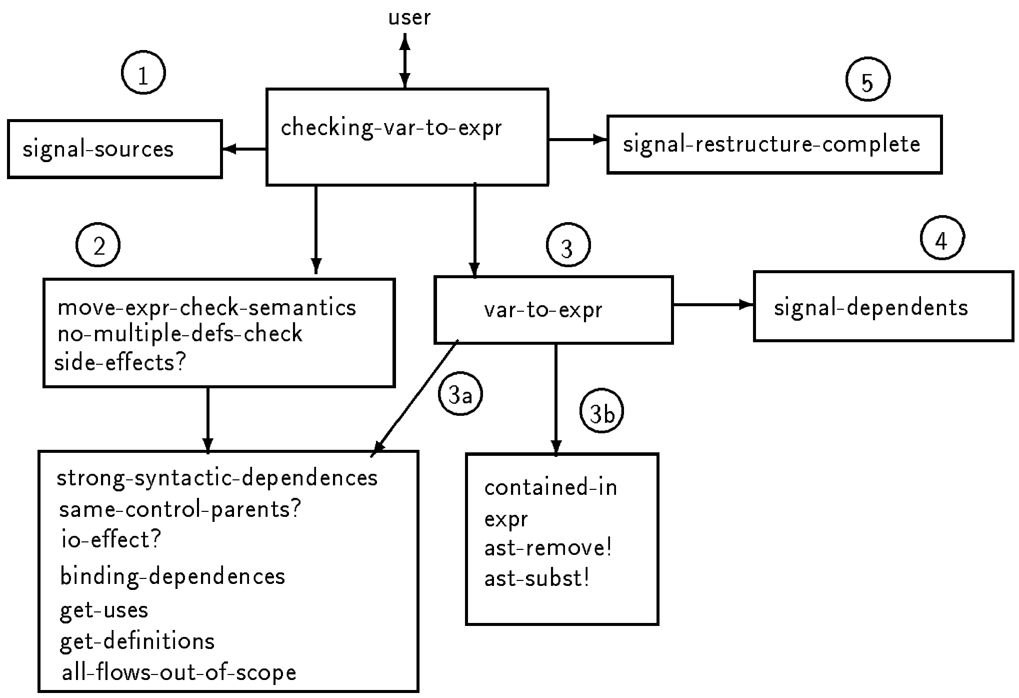


Figure 5.6: The layers and ordering within `var-to-expr`
(Arrows are function call; numbers match discussion)

the tool. The globalization equation suggests a straightforward approach to constructing transformations—manually deriving the transformation mappings and dynamically computing data mappings. The equation also explicitly obligates the transformation programmer to use the substitution rules to justify the correctness of the mapping between AST and PDG transformations.

By separately implementing the abstractions of the model, a natural separation of concerns was achieved in the implementation. In particular, separating many kinds of information such as:

- AST and PDG data representation,
- AST and PDG data consistency,

- the Flow query interface from AST and PDG representation,
- syntactic and semantic layers in the Restructuring module, and
- failure discovery and failure reporting,

made it possible to implement and modify each in isolation and to incorporate complex function without affecting other components. In practice this has made adding a new meaning-preserving transformation more straightforward than anticipated; most can be built by combining existing syntactic transformations and semantic checks. Experimenting with consistency has been possible, too, allowing incremental updates for transformations to be implemented one-by-one without requiring global change.

Likewise, enhancing the PDG to handle first-class functions, completing incremental consistency, or adding failure-driven transformations will not be complicated by unnecessarily coupled components. This separation required building extra tools to achieve the necessary integration of function. But they are small, and as the needs for integration have changed, the integrated tools have not had to change unless their function was inadequate. Because of the exploratory nature of this research, this has turned out to be a major benefit. In a non-exploratory setting this separation will still be valuable for dynamically adding and removing tools such as editors and version control.

Interestingly, the desire to keep components functionally independent allowed implementing rich component dependences. In particular, implementing component connections through relations and event-mediator integration has allowed developing and easily modifying the functions for mapping between the AST and PDG. If the components had been built together, the mappings might not have been as easy to implement. Specifically, independently implemented relations have allowed migrating from 1-1 to 1-many relations as more properties of the AST and PDG relationship needed representation.

Chapter 6

Related Work

The influence of Hoare's work [Hoare et al. 87] on the value of source-to-source transformation was discussed in the introduction, as was the influence of Parnas's work in modularization [Parnas 72] and Belady and Lehman's work on the impact of iterative maintenance on overall system cost. Likewise in Chapter 4 the usefulness of PDGs and their background has already been discussed. This chapter examines relationships to other work not yet explored. In the remaining work these relationships fall into three categories: shared goals, shared technique, and complementary goals. Few of the research projects described below fall into just one of these categories. Comparison with research that has similar goals will give a sense of the range of techniques that have been applied to reducing maintenance costs, and which are most effective. Comparison with research that shares technique will expose the general applicability of transformation to solving software problems, and expose pitfalls that have been encountered previously with this technique. Research with complementary goals addresses issues that can improve the task of tool-aided restructuring. Examining this research gives a sense of where restructuring fits in the maintenance process.

The basis for these comparisons are the essentials of the thesis research:

- reducing the cost of maintenance,

- using transformation to restructure,
- the need for human judgment in choosing a good restructuring,
- the need to preserve meaning,
- and the role of globalization in achieving that goal.

These are not examined item-by-item in each discussed work, but the appropriate points are raised as necessary.

6.1 Imposing Block Structure

A program using gotos can be automatically transformed into a program using only the structured flow graph operators *sequence*, *branch*, and *loop* [Böhm & Jacopini 66]. The intent is to improve a program's structure to lower the cost of maintenance, a shared motivation with the research of this thesis.

Most of the solutions involve simulating gotos with structured operators. One possibility is to use a large case statement inside a loop [Williams & Ossher 77]. This is done by putting each jump-free section of code in the original program into a case in the case statement. The case label can be the constant whose name is the original goto to that piece of code. Fall-through gotos are given an invented tag. Then at the end of each case a flag is set to the label of the goto that would have been jumped to next in the original program. When the top of the case is entered again on the next loop iteration, it selects the case corresponding to that label (See Figure 6.1). This proves unsatisfying when the gotos are tangled, as the result is not much prettier than the original. Some approaches try to preserve the original structure of the program during goto removal [Ramshaw 88]. These techniques use control flow graphs, an early precursor to, and important subrepresentation of, PDGs.

Automatic restructuring systems such as SUPERSTRUCTURE [Morgan 84] and RECODER [Federal Software Management Support Center 87] have successfully exploited

```
A:
  <e1>
  if <b1> then
    goto A;
  <e2>
  if <b2> then
    goto A;
```

```
const Terminate = 0;
const A = 1;
const A1 = 2;

flag := A;
while flag ~= Terminate do
  case flag of
    A:
      <e1>
      if <b1> then
        flag := A;
      else
        flag := A1;
    A1:
      <e2>
      if <b2> then
        flag := A;
      else
        flag := Terminate;
  end;
end;
```

Figure 6.1: Same program with gotos (left) and simulating gotos (right)

reorganizing program structure by removing or block-structuring gotos to aid in the maintainability of goto-laden programs. These tools are batch-oriented, avoiding the need for user input. Although this approach has shown some benefit in experiments, if the program was built with structure in mind, the system's restructuring distorted this structure and the programmers preferred the original version [Federal Software Management Support Center 87].

Although useful as a first step for programs with gotos, restructuring control is limited in usefulness. The relationships among data, functions, and types are of interest, but they are not addressed. Also, these batch techniques do not address aspects of structure that are not easy to quantify, such as restructuring towards a particular enhancement.

6.2 Transformational Programming

Introduced in the 1970's, transformational programming, also known as derivational programming, feeds a functional specification of an intended computation to a transformation system that, with guidance from a "programmer", rewrites the specification into an efficient program [Burstall & Darlington 77]. Thus most of the development effort is focused on the specification rather than programming, and there is a guarantee that the program satisfies the specification. The use of transformation is the key similarity with the restructuring work. Some basic transformations are:

- **Unfolding** substitutes a function's definition in place of its uses. Unfolding can expose identities that allow other transformations to happen, leading ultimately to its inverse, *Folding*.
- **Abstraction** substitutes a variable for every instance of an expression, and defines that variable to be the value of that expression. This exposes similarities in the code.
- **Instantiation** substitutes a value or expression for a parameter. This permits splitting a problem into cases, such as a base-case and recursive case.

The analogues of these transformations in the restructuring work are generalized to handle the semantics of imperative programming languages. Recognizing the possibility to fold has the same equivalence problem as `global-substitute-function`, and appears to use a similar pattern-matching technique.[1]

A significant detriment to transformational programming is the large number of transformations that must be applied to derive a program. This is not as severe a problem in restructuring because restructuring does not need to transform through successive language levels and also transform within a language. The transformation between language levels is essentially the choice of implementation at the lower level for the specification at

[1]The presence of referential transparency in this domain does not help. Program equivalence is undecidable due to looping, or equivalently, recursion, which is what functional languages use to iterate.

the higher level. Also, transformational programming makes more use of the algebraic knowledge of types, which although important for equivalence, do not affect locality.

Another problem is the large catalogue of transformations that must be available for deriving programs. The catalogues are large enough to make it difficult to find appropriate transformations to apply. The reason this occurs is that there are large, potentially unbounded, numbers of appropriate implementations (choice of representation and algorithm) for a specification. Local transformations must also be supported. This is not a serious problem in the restructuring domain because the catalogue is practically bounded by the size of the language's syntax.

Recent work in derivational programming has attempted to alleviate the tedium of choosing and applying transformations by building up higher-level transformations from primitives [Feather 84][Barstow 85]. For example, M.S. Feather developed a technique that uses a pattern to express the goal of a transformation [Feather 84]. Using a goal pattern, a tool can select the appropriate primitive transformations to compose to achieve the goal. Such techniques might be applicable to restructuring.

Other work has focused on trying to lower the costs of redevelopment by automating *re*derivation of a program from a modified specification [Feather 90]. The basic idea is to reuse the transformation sequence from the initial development to automate the programming tasks of maintenance. There are doubts about the success of these techniques because it appears that the ordering of transformations is brittle with respect to changes in the specification [Narayanaswamy & Cohen 91]. To avoid this problem, and the others cited above, the AP5 system uses declarative annotations that help the compiler choose transformations to derive an efficient program. This frees the programmer from transformation tasks, and an annotation needs to be changed only when the usage of the annotated program component changes. The specification and implementation are still separated, but the downside is that the specification language must be lower-level for this technique to produce efficient programs.

6.3 Knowledge Representation Enhancement

A package of tools for performing structural enhancements of a knowledge representation system [Balzer 85] has the same motivations as the research described in this thesis, but in a narrower domain. The tools exploit the highly structured, declarative domain model of a knowledge base to infer the changes to assumptions caused by a structural enhancement. A tool locates the representations that use these assumptions, so the programmer can update them. In some cases the system can perform the update as well, but lets the user provide additional input if more of a change is desired.

The changes supported are on types and attributes, and include changing the supertype of a type, or moving an attribute between types. The updates to the model are made to not only the declarations and the code that uses them, but also to the existing data modeled by the knowledge representation—an aspect not addressed in the restructuring work. Changing the type of an attribute can require translating the original values of the attribute into values of the new type. This can be automated when sufficient information about the relationship between the types is available.

It is not the intention of the tools operations to preserve meaning *per se*; the propagation of changes proceeds only one step to direct the programmer to the directly affected locations. Thus the changes in structure and the change in meaning they allow are intended to go hand-in-hand. This is in contrast to the style of restructuring described in this thesis, which propagates the changes of the compensating transformations if necessary to preserve meaning. This difference is philosophically significant, as it is widely agreed that global changes are subject to error [Parnas 72][Belady & Lehman 71], although at least with Balzer's work the user is directed to the sites potentially requiring change. It also requires significantly more work on the part of the engineer. Finally, if the user compensations are not preserving meaning, and they are global, then interfaces of the changed object are not being made any more robust; future change is not likely to be any more localized. That is, if the enhancement cannot be made locally, then the modified abstraction is poorly designed by Parnas's standards [Parnas 72] and the restructuring has not improved this, either. Although structural enhancements may not require locality of change because of the tool, functional enhancements can receive no aid from the restructuring tool, and so will require manual global change. This suggests, then, that the most reasonable action in the user-driven compensations is to make meaning-preserving compensations that localize future changes.

The semantics of type structure and attribute structure can be handled by the techniques described in this thesis, although they are not aspects that are so readily exemplified in Scheme. An area of future research is restructuring for a language with a hierarchical, extensible type structure so that reorganizing the type structure, not just subtyping, can be applied easily (see Chapter 7.7.2).

6.4 Vertical Migration and Good System Structure

One concern about "good" structure is that it can incur high execution cost overhead. John Stankovic investigated the tradeoffs between good structure and performance, developing an execution cost model based on structure, and a structuring technique called *vertical migration* to improve performance in selected portions of code [Stankovic 79][Stankovic 82]. To help identify modules that would benefit from vertical migration, Stankovic implemented an analysis tool based on the cost model. He also described a subset of transformations for improving program structure.

The cost model views a system as several layers of virtual machine, and each layer can call only the layer below it. The cost of a call from one layer into the next is broken into three parts, a *prologue* that performs some set-up or perhaps checking on entry to the layer, the execution of the function called, and an *epilogue* executed on exit from the layer. For example, calling into a layer could cross an address space boundary, requiring a context switch in the prologue and again in the epilogue when returning. This model exposes two causes for unnecessary execution overhead. One, when one layer makes multiple calls from one layer to the next, the prologue and epilogue are executed on each call. Two, if a layer wishes to call a function two layers down, it must go through the intervening layer, incurring the cost of its epilogue and prologue.

Vertical migration is the process of moving function from one layer down to the next layer, in the process removing unnecessary overhead. For example, if the layer 1 call P_{11} makes two successive calls P_{01} and P_{02} to layer 0, then a single function P_{11}' can be created in layer 0 that incorporates the two calls, and layer 2 may call it directly rather than calling P_{11}. This cuts the two calls into layer 0 down to one, and removes the call into layer 1 altogether. Additional savings are possible if the generality of the separate P_{01} and P_{02} functions is not required, and the implementation can be optimized in their absence.

One downside to vertical migration is that it violates the layering methodology by allowing layer $n+1$ to call into $n-1$ directly. Good documentation techniques can help overcome this problem. Another downside is that if generality is optimized away, then enhancements requiring this generality will require additional reimplementation effort. Of course, this is acceptable if the improvement in performance is highly desired.

Stankovic performed experiments showing that improving a system's structure through manual transformation, and then selectively migrating function to improve performance, yielded better overall structure and improved execution time in comparison to the original system.

Stankovic claims that restructuring cannot be automated because the choice of appropriate structure requires human judgment. What he failed to distinguish was the automation of transformations and the choice of what transformations are applied where. This thesis has automated the former without sacrificing human control over the latter. On the other hand, the vertical migration transformations, although meaning preserving, cannot be automated in the style presented in the thesis. This is because vertical migration eliminates execution of prologue and epilogue code, which cannot be described with the existing PDG substitution rules.

6.5 Program Understanding

Restructuring is a tool-oriented, manipulative technique for aiding maintenance. Program understanding is an analytic tool-oriented technique for aiding maintenance. Program understanding techniques—also sometimes called reverse engineering—such as graphical display of program structure [Cleveland 89], inferring abstractions [Rich & Waters 88], or assessing modularity [Schwanke 91][Embley & Woodfield 88] are used to extract program information in a more understandable or reusable form [Lewis 90].

Improving a programmer's understanding of a system makes its existing structure clearer—and hence better—just by making it better understood [Belady & Lehman 71] [Arnold 86]. Recall this is modeled by DAL in Belady and Lehman's maintenance cost equation (See Chapter 1.4.2). Although learning has limits on clarifying structure—because of the increasing amount of time required for increasingly complex programs—it requires no change to the system, which is advantageous in the short term.

More importantly, a program understanding tool can help an engineer navigate and understand a system that may need to be restructured. Also, after restructuring it can accelerate re-educating programmers about the system's new structure. This is an almost essential complement to tool-assisted restructuring to counter the potential drop in DAL.

At another level, an intelligent tool might be able automate the choices of transformations (See Section 6.2 and Chapter 7.1), or the actual desired structure. The latter will be very difficult because what constitutes "good" structure is difficult to quantify, and is dependent on future changes that are often unknown and perhaps not describable to a tool, anyway. The Programmer's Apprentice [Rich & Waters 88], a knowledge-based inferencing tool, generates plans from programming clichés that allow them to be reused in developing new code. However, the programmer still must choose when use of the plan is appropriate.

Many program understanding tools, such as slicers for debugging [Weiser 84] [Agrawal & Horgan 90], use PDGs or other flow analysis representations. This provides significant leverage for providing various tools at low cost, since they can share complex components.

6.6 Summary

The knowledge-base restructurer, the research most related to the restructuring work, takes a significant functional departure from the techniques proposed here in order to allow a wider range of compensation. The cost of the added flexibility is loss of automated meaning-preservation and error-prevention, involving the user in the compensations and debugging. It also does not help create locality of change in evolving abstractions.

The lessons from transformational programming has influenced this work. Although the restructuring domain is different, the potential problems with interactive transformation must be explored. The lesson to avoid a large transformation set has resulted in this research not focusing on local transformations. These can be handled manually with little difficulty, anyway.

The related work described throughout this thesis suggests that transformation and representations related to flow analysis are central ideas in improving programs. This commonality may allow building an environment that shares the code and data related to implementing these activities, lowering implementation costs of building transformation and analysis tools and also improving their interaction and performance.

Within this common basis there is significant variance, too. Compiler tools tend to be batch-oriented, and do not normally transform program source or transform globally. Program understanding tools are, largely, non-manipulative tools. This variance suggests that finding a common descriptive model or building an environment to share representation will be challenging.

Chapter 7

Limits and Extensions

Restructuring to aid maintenance is a broad topic. The research described thus far has been targeted to a simple language, the implemented tool is only a prototype, and extensive experiments have not been completed. There are also potentially more fundamental limits to the described research on flow analysis. Many of these limits suggest extensions to the current work; other extensions are suggested by applying this work to problems with slightly different motivations.

7.1 Low-Level Interface

Automating transformations avoids the error-prone details of restructuring, however the engineer still must recognize the relationships between the current structure and the goal structure, and then pick the transformations that will reach the goal. A higher level approach, similar to Feather's techniques [Feather 84] described in Chapter 6.2, would allow specifying a goal structure, and then tool could use the taxonomy's formal relationships between structures to choose the transformations.

Also, the interface of the current tool is command-driven rather than window-based. This is not inherent in the approach, and selection of objects and operations can be handled much more cleanly with windows and a mouse. However, the presentation of the program is still at the level of the implementation. This may show more detail than is desirable for restructuring—the tool user in many cases is thinking in terms of module structure and overall architecture [Shaw 89].

One obstruction to presenting an architectural view is the lack of high-level constructs in programming languages, which often do not go beyond the module level to describe properties such as layering and other salient relationships [Shaw 89]. Without this linguistic (i.e., syntactic) basis, it is difficult to build predictable, precise, high-level models for manipulation. As described in Chapter 2.4, the basis for restructuring transformations is the set of syntactic forms in the language. Given this basis, it is not difficult to find structurally revealing presentations of those forms that can serve as the basis for restructuring. For example, a box-and-arrow view of scope and module structure would be revealing and easy to formulate. So, to the extent that the programming language provides the basis for high-level structures, a restructuring tool can exploit them in compelling display to the tool user.

A simpler way to raise the level of the interface is to combine existing transformations into new ones. Since transformations are meaning preserving, the technical difficulties in combining them are not severe. The major impediment is that typically *all* subtransformations must succeed for the composite transformation to succeed; a transaction mechanism of some sort is required for backing out of a partially completed sequence, as describe in Chapter 5.6.

A step beyond backing out of failed composite transformations is a general undo mechanism for transformations. This is necessary for incorrectly applied transformations, or if a partially completed sequence of transformations will not be able to yield the desired structure. A way to support undo, and also backing out of partially completed transformations, is to retain prior versions of the AST and PDG, and then to allow reverting the current state to prior versions. A critical concern here is the amount of space required, since PDGs can be rather large [Ferrante et al. 87]. Recently a technique has been derived for making linked data structures retain versions [Driscoll et al. 89].

The key advantages of this result is that it applies to all data structures of a node-and-pointer variety (such as PDGs), and the amortized cost of maintaining versions is a small constant for each update to a node. As an alternative to storing data to support undo, the tool could exploit knowledge about the invertibility of applied transformations, performing undo by performing the inverses of prior operations in last-in-first-out order. This in general is possible because the transformations do not change (in particular delete) any value-changing operations that would have to be recalled through raw data storage.

7.2 Requiring Preservation of Meaning

For some kinds of changes it may not be desirable or possible to preserve meaning while restructuring. However, there are properties pertaining to the preservation of meaning that need to be preserved. For instance, changing an algorithm can make slight changes to the meaning of the program. Consider changing a hashing algorithm from open to closed hashing. Closed hashing may be faster, but will fail when the number of elements exceeds the number of hash slots.

Of course restructuring can facilitate an algorithm change by gathering the algorithm's data representation and control components into a module. But what about helping to preserve meaning (globally) while the module is being functionally modified? A potential solution would be to allow the module to be enhanced and preserve meaning for everything outside it. So when changing the algorithm, changes to flows inside the module would be allowed, but changes outside it would not. An enhanced restructuring tool could assure this by checking equivalence with respect to the subset of the program variables residing outside the algorithm's module. In a sense, the tool would be operating on a slice [Weiser 84] [Horwitz et al. 90] of the program. It is powerful because it would allow localized enhancement, but globally checks equivalence. What is lacking is any way of performing global compensation when it is detected that there is a global effect, since the intent of the programmer's change is not precisely knowable. A potential problem with this approach is keeping the PDG up to date in pace with the programmer's changes (See 7.4, below).

Another dimension to explore in breaking down the requirement for preserving meaning would be a tool that performs global enhancements—global changes that are not intended to preserve meaning. This is contrary to Parnas's claim that to reason effectively about changes requires them to be within a module. Of course, with a tool to help with the changes, this claim might no longer be valid. However, the domain of enhancements is infinite, while the domain of meaning-preserving changes is much more contained. Effectively automating activities in such an unrestricted domain is an open problem.

The notion of preserving meaning is the primary lever for globalizing transformations and preventing errors from being introduced, so it is more of an advantage than a limit, and other global program manipulation techniques will be challenged to provide the kinds of guarantees that programmers need to feel comfortable about global changes to code.

7.3 Requiring Flow Preservation

In this thesis's approach, no change to the structure is allowed to modify the flow properties of the program, even if it preserves meaning. There are two dimensions to this constraint: value-changing operations cannot be changed, and the meaning of the implementation is being preserved.

It is possible to augment flow preservation with type-oriented algebraic laws [Hoare et al. 87], which would allow reasoning not only about the identity of flows and operations, but about what operations do to flows. This is not a drastic change in the flow model. For example, a law such as $x + 0 = x$ (See Figure 7.1) for integers reduces one group of flows to another. This law would have allowed carrying restructuring one step beyond localizing `miles-rolled` in the transit example.

One difficulty with type-algebraic laws is that laws for a type created by the pro-

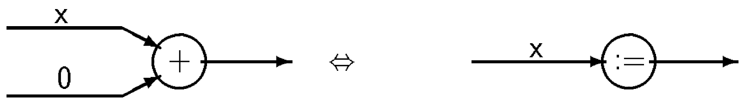


Figure 7.1: Graph representation of $x + 0 = x$

grammer are not known *a priori*. To achieve generality requires the system to derive these laws, or more practically, to allow the programmer to declare them.

The local character of many of these transformations makes them relatively easy to apply by hand, so they lack some of the benefits of automated non-local transformations. However, having them present in the tool can connect other non-local restructuring transformations without having to apply more general PDG reconstruction techniques (see the next section). Since algebraic rewrites are naturally local (even if their checks may not be), they should not interfere with keeping the PDG up to date incrementally.

A more technically difficult problem is that the *implementation* meaning of the program is being preserved with the PDG-based flow model. An implementation is just one valid interpretation of a specification, and a restructuring might be easier if it were possible to migrate to another interpretation. This might allow changing an algorithm, as discussed in the previous section. For example, choosing an arbitrary element from a set may be modeled in an implementation by retrieving the element that is quickest to access—such as the first element in a list representation. A model of meaning based on the specification of a program would be more flexible, allowing other elements to be selected with different representations. This is roughly equivalent to algebraically based restructuring in that the properties allowed by this looser model are much harder to take advantage of automatically. For example, in specifications, many properties are specified in a non-constructive, non-deterministic fashion. Determining that a change in implementation corresponds to the specification could involve a non-trivial proof.

7.4 Power and Speed of Flow Analysis

Restructuring large programs is the ultimate goal, so issues of time and space are critical. Basing the preservation of meaning on the PDG representation and our chosen implementation of it places necessary restrictions on what is practically feasible. Certain dependences, such as those introduced by procedure parameters [Shivers 88] or pointers [Larus 89], have proved costly to model precisely, and require approximate analyses that introduce spurious dependences.

Where the power of the particular flow analysis implementation falls short, the system can use other knowledge to augment the analysis. For example, the system can recall identity properties of elements it copied, and know the semantics of structures created by restructuring. Also, input from the user can help when the system is faced with spurious failure. This could take the form of answers to queries by the system when it is stuck, or annotations provided by the tool user. In this case, testing after restructuring can help find errors and improve confidence that meaning was preserved. Existing test cases would apply if restructuring occurred in the absence of functional maintenance.

The prototype currently reconstructs most of the PDG from scratch after most transformations. Chapter 4.5 describes how the substitution rules enable incremental update of the PDG during restructuring, rather than *in toto*, as the current prototype does. This is currently implemented for two of the simpler transformations, `move-expr` and `rename-variable`. The implementation of the incremental update for `move-expr` is described in Chapter 5.6. The time to execute these on a two hundred line program is several seconds rather than several minutes. With a different implementation language or more efficient representations this time would be smaller.

Space can also be a problem with PDGs. The current CL/CLOS implementation of the restructuring tool has made no effort to save space, allocating regular arrays where sparse ones may be appropriate, etc. Using another language could save significantly on space, as well. Lazily constructing portions of the PDG on demand [Venkatesh 91] could save both space and time.

If enhancements and repair are freely interleaved with restructuring, more general incremental and lazy flow analysis are necessary to increase performance substantially. There is now a significant effort in the compiler and programming environments community to incrementalize flow analysis [Burke 90] [Burke & Ryder 90] [Ryder & Paull 88].

Finally, even if poor performance remains a detriment in the future, the flow computation is precisely the kind of effort necessary to assure a given restructuring preserves meaning, but it is more time-consuming and difficult for the engineer than the computer. When the programmer makes a mistake performing this analysis the tree of change strata is augmented (See Chapter 1.4.2); protracted debugging may be required to debug the program. The computer time dedicated to accuracy is paid back in reliability and time saved in debugging.

7.5 Help Implementing Transformations

The preceding section outlined why combining existing transformations to build higher-level transformations should not be difficult. However, adding a new bottom-level transformation to the tool is not a trivial task. In particular, the tool provides no help in assuring that the transformations actually preserve meaning. The justification of the correctness is left to the tool builder, using the definition of PDGs, the commutative diagram, and the globalization equation as aids.

More formal support for showing transformations correct could depend on a theory of precisely how a text transformation in general maps to a PDG and showing that the PDG substitution rules are correct. A formal framework might enable a theorem prover to automate the proof that a transformation preserves meaning.

There has been some work on this based on denotational [Cartwright & Felleisen 89] and operational [Selke 90] semantics of PDGs, but this avenue has not been explored in this restructuring research. The work on the denotational semantics of PDGs used a staging analysis to separate a program denotation into a PDG derivation and an interpreter for the PDG. By providing a denotational language description that has a PDG intermediate form, this research has formalized the relationship between a programming language and the PDG notation.

One possible approach using denotations for designing transformations would be to modify the staged denotation of a language to allow describing transformations. Such an approach has been used to denotationally define program slices [Venkatesh 91]. Then it may be possible to use the denotational framework to show that a transformation preserves meaning, and also derive both the program and PDG versions of the transformation. (Note that even if denotational semantics could be used to prove transformations correct independent of the PDG, it would still be desirable to use the PDG abstraction as an efficient representation for performing queries.)

A *constructive* aid to building a transformations is the significant library of subtransformations and semantic checks already used in other transformations. For example, a check such as the legality of moving an expression—common to most transformations—was difficult to implement, but is reusable.

7.6 Multiple Views

Given that there is no one right structure for a program (see Chapter 1.4.1), supporting multiple, simultaneous, updatable views—structures—of a program naturally arises. Views allow providing multiple abstractions so that two agents may access the same data structure in different fashions, either for convenience or protection [Garlan 87]. For views to be effective in restructuring, the tool user must be able to make an arbitrary change to any view and have the changes propagated automatically to the other views.

This would be difficult to implement using the current restructuring model. It is not apparent that the transformations used to construct an alternative view can be incrementalized to efficiently accommodate changes to it. Also, the transformations may be brittle—small changes to the base program might require a new phrasing of the transformations to construct the same view, not unlike the problems with retransformation [Narayanaswamy & Cohen 91] (See Chapter 6.2).

As with normal restructuring, a change to an object in a view requires propagating it to alternate views. Propagation would require knowing what a change operation in one view must be in an alternate view, and what components must be compensated. Thus a first step to supporting views would be developing the technology to maintain the mapping between components in two different structurings of a program. This could be built on the techniques of event-mediator integration and relations described in Chapter 5.6. Once mapping is possible it will be necessary to figure out how to translate operations between views. This is similar to the process of translating transformations between Scheme and the PDG, as described by the commutative diagram in Chapter 4. The primary difference is that the relation between two views must be creatable dynamically, whereas the relation between the Scheme and PDG views was created statically by hand. A more mathematical basis is required to support constructing views dynamically.

7.7 Transforming Other Structural Forms

The work in this thesis has focused on transformation of the syntactic constructs of a block-structured language. For the techniques described to be widely useful requires that they be applicable to languages with modules, classes, and hierarchical type structures. The application of these techniques to programs written in functional and logic languages is not explored here, except to note that (1) similar techniques to those explored here should work, and (2) it should be easier to apply restructuring in these domains because of their algebraic transparency, which can allow direct application of transformations to the text without reference to a secondary representation of the program.

7.7.1 Modules

Modules are a natural generalization of procedures and nested scopes. A module's interface is a set of procedures and perhaps variables. Other procedures and data may be hidden below this interface. The generality of modules permits using them to finely control structure at a large scale. Since scale is a primary contributor to program complexity, this makes modules particularly valuable. The kinds of transformations that are necessary for revamping module structure are hiding a previously visible component, moving elements between modules, splitting a module or merging two modules.

Suppose a module mechanism that groups a collection of expressions and gives them a name, `module A`; allows exporting the names of the objects in the group, `export f, g;`; and allows other modules to use it, `use A`. References to external names requires the exporting module's name as a suffix, `A:g`. This is simple but represents the basic properties that need to be manipulated. Now consider some transformations on modules.

Hiding and abstraction. To hide a component that should not be accessible to external modules in its current form requires abstraction, since *some* kind of access to it must be possible if it is of any importance. For example, consider a variable exported from a module to allow access to it. The lessons of information hiding suggest that hiding this data item within its module and exporting some functions in its place is a more robust interface [Parnas 72]. Repeated function extraction can remove all the external accesses to the variable. Since it is likely that some accesses are coded more than once, `scope-subst-call` will be required to replace redundant codings with the same call.

Recall the transit example, in which the `bus` module had exported references to variable `miles-rolled`. Exposing this as a variable required some non-local changes (such as in the `tracking` module) to add ferry miles. So exporting a variable was not a good design. This problem was solved by globally extracting the `miles-rolled` references into a function into `miles-rolled`, so future changes to the representation were localized within the `bus` module. After the extraction was complete, to guarantee that the variable is truly hidden requires removing the export of the variable. This can be handled by a transformation `remove-export` that checks that the exported variable is not accessed outside the module, and then removes the export declaration.

Supporting inter-module function extraction would require a small extension to the current `extract-function` transformation. In conjunction with creating the function, an export of the function must be generated so it is visible at the point of extraction. This is a new contour operation, possible (and necessary) because modules do not explicitly export symbols the way nested scopes do. To continue using the transit example, consider when `total-miles` is extracted from references to `rolled-miles`. The tool must generate an `export total-miles` in the `bus` module so `total-miles` is visible in the `tracking` module. No modification of imports is necessary because the presence of the extracted code in `transit` means that `bus` already must be imported.

Move. Moving an element between modules (say moving `f` between modules `A` and `B`) can be invoked by the engineer as `move-expr` is in the current tool, but some extra tool action is required. In particular if the element is exported, and used by other modules, it is necessary for the tool to check that each uses module `B`, or that the necessary `use B` declarations are added. These are non-local checks and changes. Also, if `f` is the only element accessed by a module, it may be desirable to remove the `use A` declaration. Since these alternatives have different structural results, different transformations need to be implemented for the different choices. Alternatively the tool can interact with the tool user so that it makes only the changes that the user desires.

Split. Splitting a module is like creating a new empty module and moving several elements into it, except here it is more likely that the intent is to modify `use` declarations extensively.

Merge. Again, this is like moving, except that all of the components of one module are moved into another, with substantial reworking of `use` declarations.

Although these transformations are simple, they are valuable because of the scale and volume at which they may be applied. The distance between the non-local updates, in terms of the number lines of code or the number of files can be quite large. Also, the number of non-local updates and the scope conflicts in a single move may be large.

7.7.2 Class Hierarchies

The key property of a class hierarchy is the sharing of implementation and the polymorphism that results from the overlapping of shared interfaces. In particular, if an object is seeking a method to invoke then it looks in the class hierarchy: first in the class of the object that the method is being invoked on, and then in its superclass, and so forth until the desired method is found or the root is reached. This upward tree search is a structurally rich metaphor, and any manipulation of the tree that preserves the properties of the search will preserve meaning.

During development and enhancement, a class hierarchy can be used as a mechanism to add function without changing existing code. The result may be that the hierarchy represents the implementation history of the program. However, it is generally desirable for the hierarchy to represent the type structure of the application domain, in spite of this tendency [Johnson & Foote 88]. Automated restructuring can help manage the tension between the two uses [Opdyke & Johnson 90]. Also, discovery of design errors may suggest changing the interface of a superclass method, resulting in restructuring. Such a change would propagate to subclass definitions of the method and to all uses of the methods. Such migration and repair requires transformations to merge subclasses with superclasses, and perhaps moving function out of superclasses into subclasses, as well as the generalized versions of transformations already discussed for pascal-like languages.

A significant challenge to supporting restructuring for a class-based language is the effect that polymorphism has on the control-flow graph. Since at any call point any number of methods with the same name may be denoted, the branching in the graph is significant, and so any transformation must satisfy the constraints of all those methods, not just one. Fortunately, type and data flow information can be used to refine the control-flow graph significantly [Chambers & Ungar 90], eliminating much of the branching effect. Only actual experience will reveal, however, if this will be sufficient.

The class hierarchy can complicate the actual transformations, also. In particular, any manipulation of a class method may imply similar changes to every subclass redefinition of that method.

Extracting a method. As an example of this additional complexity, consider a transformation for extracting a method from inline code. Consider the program fragment below, which defines a class method `ratio` for class `c`, and one also for `c`'s subclass `s`. Suppose the engineer decides that the constant `0.5` in `c::ratio` needs to be extracted into a new method `c::factor`.[1] A natural implication is that this new method should return `0.5` for calls inside `c::ratio`, but `0.4` inside `s::ratio`. In other words, multiple methods need to be extracted.

```
(* superclass method *)
method c::ratio ()
  return self.x * 0.5;
end

(* subclass method *)
method s::ratio ()
  return self.x * 0.4;
end
        .
        .
        .
        .

    obj.ratio();        (* a call to method ratio *)
```

One way this can be handled is to generalize `extract-function` to transformation `extract-method`, which is capable of extracting a method for the initial extraction, and also others for subclasses. In the example, the context of the initial extraction is `c::ratio`, so in subclass `s` the implied context for extraction is `s::ratio`. Finding the embedded expression to extract in this definition is not necessarily trivial. In fact, this is a more general case of the program equivalence problem described in Chapter 2.4 for transformation `scope-sub-call`. Similar solutions to those in `scope-sub-call` can be

[1]A variation of this example and the ones following were suggested in the context of describing transformations that can be used to assure a program obeys the *Law of Demeter* [Lieberherr et al. 88].

used, such as using the output type or the usage of the initially selected expression (`0.5`) to narrow the possibilities. These would work for the example above, but since the algorithm is a heuristic, final approval of the tool's choice by the engineer is necessary. The result, after the engineer approves the tool's additional extraction of `0.4` for `s::factor`, is:

```
(* superclass method *)
method c::ratio ()
  return self.x * self.factor();
end

method c::factor ()
  return 0.5;
end

(* subclass method *)
method s::ratio ()
  return self.x * self.factor();
end

method s::factor ()
  return 0.4;
end
        .
        .
        .

    obj.ratio();        (* a call to method ratio *)
```

`extract-method` has the same structural benefit as `extract-function`—introducing abstraction to hide details—but the presence of the hierarchy allows compensating multiple definitions as well as multiple uses, increasing the non-local character of the transformation. The multiple definition compensation also makes the resulting abstraction more complete: in the `ratio` example not only was the embedded constant removed in *both* methods, but both constants were merged into the *same* abstraction.

Adding a method parameter. Now consider generalizing the interface of a method that is defined in a superclass and redefined in subclasses. This is like the transformation `expr-to-binding` described in Chapter 2.4, but like `extract-method` requires doing the abstraction on more than one definition, and in a consistent fashion.

Starting with the same initial program fragment as in the previous example, it is desired to abstract the factor `0.5` from `c::ratio`. It is implied that the subclass method `s::ratio` must be similarly transformed for the interfaces to remain syntactically consistent.

As with method extraction, since the two definitions of the `ratio` do not have the same implementation, finding the embedded expression to extract in each definition is complicated. Worse, since either definition of `ratio` may be invoked in the call at the end of the fragment, neither one or the other abstracted expression can be put in at the call site if meaning is to be preserved. One heuristic is for the tool to ask the user if one expression subsumes the other, and to use that as the abstracted value for all of them. Another is for the tool to use type inference to determine if there is only one method that is really callable at the site.

A simpler solution is for the engineer to first apply the method extraction used in the previous example to avoid the problem:

```
(* superclass method *)
method c::ratio ()
  return self.x * self.factor();
end

method c::factor ()
  return 0.5;
end

(* subclass method *)
method s::ratio ()
  return self.x * self.factor();
end

method s::factor ()
  return 0.4;
```

```
end
        .
        .
        .

    obj.ratio();        (* a call to method ratio *)
```

Now the expressions to be extracted are identical, so `self.factor` may be successfully abstracted and substituted in the call:

```
(* superclass method *)
method c::ratio (fact)
  return self.x * fact;
end

method c::factor ()
  return 0.5;
end

(* subclass method *)
method s::ratio ()
  return self.x * fact;
end

method s::factor ()
  return 0.4;
end
        .
        .
        .

    obj.ratio(obj.factor());        (* call to generalized ratio *)
```

The bottom line is that the class version of this transformation is virtually identical to the original, but as with method extraction the compensations involve multiple definitions, as well as multiple uses, increasing the non-local character of the transformation. Unlike method extraction, however, the change is in the interface of the method, so the compensations are *necessary* for the program to remain syntactically legal.

Popping. This is moving a method (or instance variable) up from a class to its superclass. This presents little problem unless the method already exists in the superclass, or in sibling classes that may need to be moved up, too. Here, as with global function extraction, it is necessary to recognize equivalently coded functions to see if they can be successfully merged in the superclass.

One possible conflict for popping a method is that each subclass version of the method has an embedded constant in its version that is different from the others [Lieberherr et al. 88]. Consider for example a method `ratio` implemented in two subclasses, one embedding a reference to `0.5` the other `0.4`:

```
method s1::ratio ()                    method s2::ratio ()
  return self.x * 0.5;                   return self.x * 0.4;
end                                    end
        .
        .
        .

    obj.ratio();        (* a call to method ratio *)
```

where `s1` and `s2` are subclasses of class `c`. To allow popping requires three steps. The first hides the `ratio` call in another subclass method via `extract-method`, which can apply `extract-function` multiple times on subclass methods:

```
method s1::ratio ()                    method s2::ratio ()
  return self.x * 0.5;                   return self.x * 0.4;
end                                    end


method s1::myratio ()                  method s2::myratio ()
  return self.ratio();                   return self.ratio();
end                                    end
        .
        .
        .

    obj.myratio();        (* call modified to call myratio *)
```

The second step abstracts the constant in each `ratio` method implementation as a parameter via `expr-to-binding` to create a more general interface:

```
method s1::ratio (factor)              method s2::ratio (factor)
  return self.x * factor;                return self.x * factor;
end                                    end


method s1::myratio ()                  method s2::myratio ()
  return self.ratio(0.5);                return self.ratio(0.4);
end                                    end
        .
        .
        .

    obj.myratio();        (* call not affected by abstraction *)
```

After this the `ratio` methods are equivalent, and may be popped up to `c`:

```
method c::ratio (factor)
  return self.x * factor;
end

method s1::myratio ()                  method s2::myratio ()
  return self.ratio(0.5);                   return self.ratio(0.4);
end                                    end
        .
        .
        .

    obj.myratio();        (* call not affected by pop *)
```

Pushing. Pushing is moving a method or instance down the hierarchy. This is complicated because, for example, it must be verified that a superclass object never actually accesses the method to be moved (unless it is just copied down, rather than moved). Since subclass objects can often be used in a superclass variable. Flow analysis in the PDG can reveal if the usage of the method allows the movement.

In contrast to popping, pushing a method may need to *instantiate* a parameter as an embedded constant. Instantiation is handled by applying in reverse order the inverses of the abstracting transformations, first the push, followed by `binding-to-expr` and `inline-method`.

By applying multiple pops and pushes many other operations can be implemented, such as spreading or merging sub-behavior between subclasses, or merging an entire subclass with a superclass.

Renaming. Unlike other language structures, classes hierarchies allow many procedures to be denoted by the same name, and that name may denote any of them in a call, depending on runtime behavior. Thus to change the name of a method really means, potentially, changing the name of many methods. Although it is not difficult to implement such a transformation, it is highly non-local in that all the uses *and* definitions must be found and updated, with assurance that the change does not affect any existing method protocols.

Chapter 8

Conclusion

Maintenance is the most expensive component of the software process [Lientz & Swanson 80]. The structure of a program significantly influences the cost of its maintenance. Restructuring a program can isolate a design decision in a module so that changing it will not require costly non-local changes. Automating the non-local activities of restructuring to make it cost-effective shows promise. In particular, this approach allows the software engineer to locally specify source-to-source structural changes, while automating the non-local, consistent changes that complete the restructuring. This accelerates restructuring and avoids introducing new errors due to inconsistencies in the changes.

8.1 Critique

This thesis has shown how to build an interactive, source-to-source, meaning-preserving restructuring tool for imperative programs that allows the engineer to locally specify a global structural change. By successfully following the model, the implementation credits it as a framework for understanding the relationships among the non-local changes of a transformation. The prototype also provided a concrete basis for evaluating manual restructuring techniques in the experiment. The experiment provided evidence that automating restructuring should have concrete benefits by automating activities that tend to be haphazard when performed manually. Restructuring, however, is not yet a mature technology:

A more formal model for maintenance and restructuring is needed. Belady and Lehman's work crudely predicts that automated restructuring has measurable benefits by reducing structural complexity for individual changes. A more accurate analysis requires a micro-model of software change. Experiments in the near future will explore a programmer's tendency to restructure to ease change, and what factors influence the decision (See Chapter 3.3). These experiments will not require significant changes to the tool, and will provide early experience for developing a micro-model.

A more formal model for transformation is required. Program dependence graphs and their formal relation to programs via the commutative diagram supports the hypothesis that a meaning-preserving restructuring tool can be built with reasonable performance and reliability. The method of globalization to create program transformations from PDG transformations is effective but informal. A more formal relation between transformations on a PDG and transformations on a program would add strength to this result. As discussed in Chapter 7.5, techniques exploiting recent work on the semantics of PDGs seem promising [Cartwright & Felleisen 89][Selke 90][Venkatesh 91].

Transformational restructuring is low-level. After the software engineer chooses the best structure for the next maintenance step, the transformational approach still requires the engineer to choose the transformations that migrate the program from its existing structure to the new structure. Chapter 7.1 suggested several techniques for countering these problems, perhaps the most promising being a goal-directed approach to program derivation [Feather 84], which for restructuring would allow the engineer to specify the goal structure and let the system infer the transformations.

The relationship between restructuring and preserving meaning is unclear. The preservation of meaning is a central theme of this restructuring work. Three aspects of this may deserve further attention. First, is preserving meaning needed as an automation technique for restructuring? Balzer's tool described in Chapter 6.3 leads the engineer to the locations requiring update and performs some manipulations, but does not preserve meaning. This is more flexible than this thesis's approach, but less automatic and provides fewer guarantees. Which approach will engineers prefer? The experiment in Chapter 3 suggests that a guarantee of preserved meaning is valuable, but the subjects were unaided except for a text editor. Balzer's writings provide no information on this. Second, if meaning is to be preserved, must it be the *implementation* meaning of the program? For example, an approach based on preserving the specification meaning would allow more flexibility. Discussed briefly in Chapter 7.3, the primary concern is that specification semantics may allow too much flexibility to allow automated restructuring. Third, is preserving meaning during restructuring too restrictive to be useful? This reflects on the other two points, but also questions whether global enhancement must be automated to successfully reduce maintenance costs. This calls Parnas's module work into question, but the addition of a tool to automate non-local enhancement may qualitatively change the basis of his assumptions.

Experience must establish that automated restructuring is cost-effective. The experiment in Chapter 3 showed restructuring was haphazard and error-prone when performed manually. Although this experiment was on a small program—where no benefit might be seen—the differences were observable. Likewise, the restructurings of matrix multiply (Chapter 2.5) and the KWIC indexing program (Appendix A) using the prototype have strengthened the case for tool-aided restructuring. Certainly more thorough experiments are required, but only long-term use will provide sure evidence of the value of tool-aided restructuring. In either case, interactive performance, a window interface, and a more complete and robust set of transformations are required for further progress.

Restructuring must be concretely generalized to handle large programs. The greatest potential for restructuring lies in managing the structure of large programs. At the scale of thousands or millions of lines of code, the asymptotic term of exponential structural complexity of a program will dominate, with great financial impact. Modules and classes are two language structures that are useful for building large programs.

Chapter 7.7 introduced transformations for restructuring at the module level and in a class-hierarchy. These transformations need to be incorporated in a tool, justified in the model, and their success measured in use. Also, the ability to incrementally update the tool's underlying PDG representation using the substitution rules promises good performance for large-scale restructuring. Increasing the scale will test this claim, environment issues such as storage between restructuring sessions, the power of the individual transformations, and the ability to transform in the presence of interdependences derived from conservative dependence analysis. The techniques discussed in Chapter 7.4 will be important here.

More significantly, manual restructuring at a large scale becomes untenable, with so many non-local relationships to keep consistent during change. The manual bookkeeping promises to be overwhelming, so only an automating tool can flawlessly and tirelessly restructure with a guarantee of preserving meaning.

8.2 Contributions

This thesis is a first step towards demonstrating that tool-aided restructuring can lower the costs of maintenance. Specifically, this thesis has shown how to build an interactive, source-to-source, meaning-preserving program restructuring tool for imperative programs that allows the engineer to locally specify a non-local change. The transformations do not just remove gotos; the technique supports a broad class of transformations for localizing design decisions. Further, this thesis has proven the novel idea that program structure can be managed by transforming the abstractions of a program without affecting its basic computations. There are several supporting contributions:

Using meaning-preserving transformation to automate restructuring. The macro-model of Belady and Lehman [Belady & Lehman 71] predicts that manual restructuring will be as costly as any other maintenance activity, primarily because it requires physically distributed, but semantically consistent, changes to the program. Also, the experiment in Chapter 3—observing the manual restructuring of a matrix multiply program—revealed that manual restructuring is haphazard and error-prone.

Automating restructuring is enabled by requiring that structural changes not change the runtime behavior of the program: preserving meaning is a precise, easy to understand, global consistency constraint that does not permit introducing errors during restructuring, and still allows changing structure. Using a transformational approach, Chapter 2 defined a set of structural changes exploiting this constraint. The transformations as a group can manipulate structures spanning several common types.

The transformations are not new [Loveman 77][Burstall & Darlington 77][Stankovic 82], but their style of application and purpose are. A transformation is applied to a single syntactic construct, and the tool can make the compensating changes in the rest of the program to preserve its original meaning. This style of transformation removes the engineer from error-prone activities without sacrificing control over the resulting structure, unlike prior restructuring tools [Federal Software Management Support Center 87] [Morgan 84].

In addition to revealing the hazards of manual restructuring, the experiment also confirmed that the tool automates those exact activities that are error-prone: making consistent, physically distributed changes. It also showed that the tool's style is consistent with observed manual restructuring techniques, which should improve usability. In particular, the common manual technique of applying copy-paste to a construct, and then editing it and its uses is supported by the tool. The tool allows the engineer to apply the first part of the action, with the tool completing the editing and compensations to the uses.

Development of a practical model for defining meaning-preserving source-to-source transformations. The program dependence graph (PDG) provides exactly the semantic information necessary to reason about non-local program changes in a local fashion. However, the PDG lacks information on scopes, which is represented in the program text. The model introduced in Chapter 4 defined a small set of local, meaning-preserving subgraph substitution rules and scope manipulation rules. By relating the PDG to the program source via the mappings of a commutative diagram, the rule set locally describes a transformation's physically distributed textual changes. This simplified understanding the meaning-preserving properties of program transformations.

The globalization equation, derived from the commutative diagram, provided guidance for mapping PDG transformations to program transformations during implementation. Only searches are dynamically mapped between the program and the PDG; all updates to the program and PDG can be performed directly on each. Consequently, a source-to-source transformation is applied directly to the program, leaving unchanged those syntactic features of the program that are not explicitly manipulated. This is in contrast to standard PDG unparse techniques [Horwitz et al. 90] [Larus 89].

A working implementation of a restructuring tool. Successfully implementing a restructuring tool validated the claim that a restructuring transformation can be invoked locally by the engineer and compensated by a tool to preserve meaning. The tool was successfully used on two programs: the matrix multiply used in the experiment, and the inadequate control decomposition of KWIC. The implementation also supports the claim that the model is a powerful tool. In particular, the model's abstractions and the globalization equation helped define the tool structure and reason about the correctness of its transformations. The implementation also tested incremental update of the PDG in two instances, evidence that a restructuring tool can be efficient.

Bibliography

[Agrawal & Horgan 90] H. Agrawal and J. R. Horgan. Dynamic program slicing. In *Proceedings of the SIGPLAN '90 Conference on Programming Languages Design and Implementation*, June 1990. SIGPLAN Notices 25(6).

[Aho et al. 86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers, Principles, Techniques, and Tools*. Addison-Wesley, Reading, Mass., 1986.

[Allen & Cocke 72] F. E. Allen and J. Cocke. A catalogue of optimizing transformations. In R. Rustin, editor, *Design and Optimization of Compilers*. Prentice-Hall, Englewood Cliffs, NJ, 1972.

[Arnold 86] R. S. Arnold. An introduction to software restructuring. In R. S. Arnold, editor, *Tutorial on Software Restructuring*. Society Press (IEEE), Washington D.C., 1986.

[Balzer 85] R. Balzer. Automated enhancement of knowledge representations. In *Proceedings of the Ninth International Joint Conference Artificial Intelligence*, pages 203–207, August 1985.

[Barstow 85] D. Barstow. On convergence toward a database of program transformations. *ACM Transactions on Programming Languages and Systems*, 7(1):1–9, January 1985.

[Belady & Lehman 71] L. A. Belady and M. M. Lehman. Programming system dynamics or the metadynamics of systems in maintenance and growth. Research Report RC3546, IBM, 1971. Page citations from reprint in M. M. Lehman, L. A. Belady, editors, *Program Evolution: Processes of Software Change*, Ch. 5, APIC Studies in Data Processing No. 27. Academic Press, London, 1985.

[Belady & Lehman 76a] L. A. Belady and M. M. Lehman. A model of large program development. *IBM Systems Journal*, 15(3):225–252, 1976. Reprinted in M. M. Lehman, L. A. Belady, editors, *Program Evolution: Processes of Software Change*, Ch. 8, APIC Studies in Data Processing No. 27. Academic Press, London, 1985.

[Belady & Lehman 76b] L. A. Belady and M. M. Lehman. Program evolution and its impact on software engineering. In *Proceedings of the 2nd International Conference on Software Engineering*, pages 350–357, October 1976. Reprinted in M. M. Lehman, L. A. Belady, editors, *Program Evolution: Processes of Software Change*, Ch. 9, APIC Studies in Data Processing No. 27. Academic Press, London, 1985.

[Boehm 81] B. W. Boehm. *Software Engineering Economics.* Prentice-Hall, Englewood Cliffs, NJ, 1981.

[Böhm & Jacopini 66] C. Böhm and G. Jacopini. Flow diagrams, Turing machines and languages with only two formation rules. *Communications of the ACM*, 9(5):366–71, May 1966.

[Burke & Ryder 90] M. Burke and B. G. Ryder. A critical analysis of incremental iterative data-flow algorithms. *IEEE Transactions on Software Engineering*, SE-16(7), July 1990.

[Burke 90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[Burstall & Darlington 77] R. M. Burstall and J. Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Callahan et al. 90] D. Callahan, S. Carr, and K. Kennedy. Improving register allocation for subscripted variables. In *Proceedings of the SIGPLAN '90 Conference on Programming Languages Design and Implementation*, June 1990. SIGPLAN Notices 25(6).

[Cartwright & Felleisen 89] R. Cartwright and M. Felleisen. The semantics of program dependence. In *Proceedings of the SIGPLAN '89 Conference on Programming Languages Design and Implementation*, July 1989. SIGPLAN Notices 24(7).

[Chambers & Ungar 90] C. Chambers and D. Ungar. Iterative type analysis and extended message splitting: Optimizing dynamically-typed object-oriented programs. In *Proceedings of the SIGPLAN '90 Conference on Programming Languages Design and Implementation*, June 1990. SIGPLAN Notices 25(6).

[Cleveland 89] L. Cleveland. A program understanding support environment. *IBM Systems Journal*, 28(2), 1989.

[Collofello & Buck 87] J. S. Collofello and J. J. Buck. Software quality assurance for maintenance. *IEEE Computer*, pages 46–51, September 1987.

[Cunnington et al. 90] W. Cunnington, R. Johnson, M. Linton, and TBD. Designing reusable designs—experiences designing object-oriented frameworks. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, page 234, October 1990. Panel Discussion, Allen Wirfs-Brock, moderator. SIGPLAN Notices 25(10).

[Cytron et al. 88] R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and K. Zadeck. An efficient method of computing static single assignment form. In *Proceedings of the 16th Symposium on Principles of Programming Languages*, pages 25–35, January 1988.

[DeMillo et al. 79] R. DeMillo, R. Lipton, and A. Perlis. Social processes and proofs of theorems and programs. *Communications of the ACM*, 22(5), May 1979.

[Dijkstra 72] E. Dijkstra. *Notes on Structured Programming*, pages 1–82. Academic Press, New York, 1972.

[Downey & Sethi 78] P. J. Downey and R. Sethi. Assignment commands with array references. *Journal of the ACM*, 25(4):652–666, October 1978.

[Driscoll et al. 89] J. R. Driscoll, N. Sarnak, D. D. Sleator, and R. E. Tarjan. Making data structures persistent. *Journal of Computer and System Science*, 38(1):86–124, February 1989.

[Dybvig 87] R. K. Dybvig. *The Scheme Programming Language*. Prentice-Hall, Englewood Cliffs, NJ, 1987.

[Embley & Woodfield 88] D. W. Embley and S. N. Woodfield. Assessing the quality of abtract data types written in Ada. In *Proceedings of the 10th International Conference on Software Engineering*, pages 144–153, April 1988.

[Feather 84] M. S. Feather. A system for assisting program transformation. *ACM Transactions on Programming Languages and Systems*, 4(1):1–20, January 1984.

[Feather 90] M. S. Feather. Specification evolution and program (re)transformation. In *Proceedings of the 5th RADC Knowledge-Based Software Assistant Conference*, September 1990.

[Federal Software Management Support Center 87] Federal Software Management Support Center. Parallel test and productivity evaluation of a commercially supplied COBOL restructuring tool. Technical report, Office of Software Development and Information Technology, September 1987.

[Feldman 79] S. Feldman. Make—a program for maintaining computer programs. *Software Practice and Experience*, 9(4), April 1979.

[Ferrante et al. 87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.

[Garlan 87] D. Garlan. *Views for Tools in Integrated Programming Environments.* PhD dissertation, Carnegie-Mellon University, Department of Computer Science, 1987.

[Hoare et al. 87] C. A. R. Hoare, I. J. Hayes, H. Jifeng, C. C. Morgan, A. W. Roscoe, J. W. Sanders, I. H. Sorensen, J. M. Spivey, and B. A. Sufrin. Laws of programming. *Communications of the ACM*, 30(2):672–686, August 1987.

[Horwitz et al. 88] S. Horwitz, J. Prins, and T. Reps. On the adequacy of program dependence graphs for representing programs. In *Proceedings of the 15th Symposium on Principles of Programming Languages*, pages 146–157, January 1988.

[Horwitz et al. 89] S. Horwitz, J. Prins, and T. Reps. Integrating noninterfering versions of programs. *ACM Transactions on Programming Languages and Systems*, pages 345–387, July 1989.

[Horwitz et al. 90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.

[Johnson & Foote 88] R. E. Johnson and B. Foote. Designing reusable classes. *Journal of Object Oriented Programming*, pages 22–35, June/July 1988.

[Kuck et al. 81] D. J. Kuck, R. H. Kuhn, B. Leasure, D. A. Padua, and M. Wolfe. Dependence graphs and compiler optimizations. In *Proceedings of the 8th Symposium on Principles of Programming Languages*, pages 207–218, January 1981.

[Larson 88] P. Larson. Dynamic hash tables. *Communications of the ACM*, 31(4), April 1988.

[Larus & Hilfinger 88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Languages Design and Implementation*, June 1988. *SIGPLAN Notices*, 23(7).

[Larus 89] J. R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD dissertation, UC Berkeley Computer Science, May 1989. Also Technical Report No. UCB/CSD 89/502.

[Lehman 80] M. M. Lehman. On understanding laws, evolution and conservation in the large-program life cycle. *J. Systems and Software*, 1(3), 1980. Page citations from reprint in M. M. Lehman, L. A. Belady, editors, *Program Evolution: Processes of*

Software Change, Ch. 18, APIC Studies in Data Processing No. 27. Academic Press, London, 1985.

[Lewis 90] T. Lewis, editor. *IEEE Computer.* IEEE Computer Society, January 1990. Special Issue on Software Engineering, W. M. Osborne, E. J. Chikofsky special Eds.

[Lieberherr et al. 88] K. Lieberherr, I. Holland, and A. Riel. Object-oriented programming: an objective sense of style. In *Proceedings of the Conference on Object Oriented Programming Systems, Languages and Applications*, pages 323–334, November 1988. SIGPLAN Notices 23(11).

[Lientz & Swanson 80] B. Lientz and E. Swanson. *Software Maintenance Management: A Study of the Maintenance of Computer Application Software in 487 Data Processing Organizations.* Addison-Wesley, Reading, MA, 1980.

[Loveman 77] D. B. Loveman. Program improvement by source-to-source transformation. *Journal of the ACM*, 24(1):121–145, January 1977.

[Morgan 84] H. W. Morgan. Evolution of a software maintenance tool. In *Proceedings of the Second National Conference EDP Software Maintenance*, pages 268–278, 1984.

[Narayanaswamy & Cohen 91] K. Narayanaswamy and D. Cohen. An assessment of the AP5 programming language - theory and experience. Technical report, Information Sciences Institute, University of Southern California, Los Angeles, CA, March 1991.

[Opdyke & Johnson 90] W. F. Opdyke and R. E. Johnson. Refactoring: An aid in designing application frameworks and evolving object-oriented systems. In *Proceedings of the 1990 Symposium on Object-Oriented Programming Emphasizing Practical Applications*, pages 274–282, September 1990.

[Ossher 87] H. L. Ossher. A mechanism for specifying the structure of large, layered programs. In B. Shriver and P. Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 219–252. MIT Press, Boston, 1987.

[Ottenstein & Ottenstein 84] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, April 1984.

[Parnas 72] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–58, December 1972.

[Parnas 79] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Transactions on Software Engineering*, SE-5(2):128–138, March 1979.

[Perlis 88] A. Perlis, Summer 1988. Talk at IBM Almaden Research, San Jose, CA.

[Podgurski & Clarke 90] A. Podgurski and L. A. Clarke. A formal model of program dependences and its implications for software testing, debugging, and maintenance. *IEEE Transactions on Software Engineering*, SE-16(9):965–979, 1990.

[Ramshaw 88] L. Ramshaw. Eliminating go to's while preserving program structure. *Journal of the ACM*, 35(4):893–920, October 1988.

[Rich & Waters 88] C. Rich and R. C. Waters. The programmer's apprentice: A research overview. *IEEE Computer*, pages 11–25, November 1988.

[Ryder & Paull 88] B. G. Ryder and M. C. Paull. Incremental data-flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

[Schwanke 91] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *Proceedings of the 13th International Conference on Software Engineering*, pages 83–92, May 1991.

[Selby & Basili 91] R. W. Selby and V. R. Basili. Analyzing error-prone system structure. *IEEE Transactions on Software Engineering*, SE-17(2):141–152, February 1991.

[Selke 90] R. P. Selke. Transforming program dependence graphs. Technical Report TR90-131, Department of Computer Science, Rice University, Houston, Texas, 1990.

[Shaw 89] M. Shaw. Larger scale systems require higher-level abstractions. In *Proceedings of the Fifth International Workshop on Software Specification and Design*, pages 143–146. IEEE Computer Society, 1989. ACM SIGSOFT Software Engineering Notes, 14(3).

[Shivers 88] O. Shivers. Control flow analysis in Scheme. In *Proceedings of the SIGPLAN '88 Conference on Programming Languages Design and Implementation*, pages 164–174, July 1988. SIGPLAN Notices 23(7).

[Stankovic 79] J. A. Stankovic. *Structured Systems and Their Performance Improvement through Vertical Migration.* PhD dissertation, Brown University, May 1979. Dept. of Computer Science Technical Report CS-41.

[Stankovic 82] J. Stankovic. Good system structure features: Their complexity and execution time cost. *IEEE Transactions on Software Engineering*, SE-8(4):306–318, July 1982.

[Stevens et al. 74] W. Stevens, G. Myers, and L. Constantine. Structured design. *IBM Systems Journal*, 13(2), 1974.

[Sullivan & Notkin 90] K. J. Sullivan and D. Notkin. Reconciling environment integration and component independence. In *Proceedings of the SIGSOFT '90 Fourth Symposium on Software Development Environments*, December 1990.

[Tichy 82] W. F. Tichy. Design, implementation, and evaluation of a revision control system. In *Proceedings of the 6th International Conference on Software Engineering*, September 1982.

[Venkatesh 91] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the SIGPLAN '91 Conference on Programming Languages Design and Implementation*, June 1991. SIGPLAN Notices 26(6).

[Weiser 84] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Williams & Ossher 77] M. H. Williams and H. L. Ossher. Conversion of unstructured flow diagrams to structured form. *Computer Journal*, 21(2):161–167, 1977.

[Yang 90] W. Yang. *A New Algorithm for Semantics-Based Program Integration.* PhD dissertation, University of Wisconsin, August 1990. Computer Sciences Technical Report No. 962.

[Yang et al. 89] W. Yang, S. Horwitz, and T. Reps. Detecting program components with equivalent behaviors. Technical Report 840, Computer Sciences Department, University of Wisconsin, Madison WI, April 1989.

Appendix A

Restructuring Parnas's KWIC

One significant restructuring has been shown in the body of the thesis. Presented here is a second example, the KWIC indexing program, which is larger, uses a broader range of transformations, and whose possible structures are familiar to much of the software engineering community. The KWIC program takes a list of lines of text, and produces a list of all circular shifts of those lines, in sorted order.

Parnas used the KWIC indexing program to demonstrate the principle of information hiding as a criterion for module decomposition [Parnas 72]. In particular, the example showed that modules should be chosen to isolate design decisions that are likely to change in the future, and that typically these revolve around the choice of data representation. As in Parnas's comparison, the modules are not enforced by the language, but are defined by how programming tasks are assigned to individuals. Of course, with this simple example a single individual can code the whole program with little trouble, but the example is treated as though each module were assigned to a separate group or individual to implement. Thus the primary concern is whether a desired change can be made by a single group responsible for a single module. If not, the cost will be higher than otherwise because of the communication necessary between groups to coordinate the change.

Loosely, the two modularizations that Parnas chose to compare were a *functional* decomposition of work assignments, and a *data* decomposition. The functional decomposition had the following modules: (1) An input module for reading the lines of input and storing them in a data structure; (2) a circular shift module for performing the circular shifts of the original input; (3) an alphabetization module for ordering the shifts; (4) an output module for writing out the alphabetized shifts; and (5) a master control module for ordering the execution of the other modules. A plausible Scheme implementation of this decomposition is shown in Section A.3. It has one function for each module, with the necessary sub-functions implemented as local functions. Data is shared between the modules by global data structures.

For the data decomposition, Parnas chose the following module structure: (1) A line-storage module, which supports storing and retrieving individual lines of text (by line number), as well as routines for reporting the number of lines in the store, words in a line, and words in the store; (2) an input module, which calls the line-storage module to store the lines of its input; (3) a circular shifter module, which retrieves circular shifts of (by shift number, counting from the beginning of the line-storage), but also has a set-up routine that must be called before any shifts are retrieved; (4) an alphabetizing module that supports a function retrieving the shifted lines (again, by numerical index) in sorted order, and like the shifter, requires calling a set-up function first; (5) an output module that uses the alphabetizer's retrieval function to write out the sorted shifts; and (6) a master control module for ordering the execution of the other modules. A plausible Scheme implementation of this decomposition is shown in Section A.4. Basically, this program organizes modules around data structures, each module using a small set of functions to abstract away the representation.

Parnas emphasizes that the two decompositions can share all representation choices and access methods—only what is isolated in a module need be different. The program in Section A.3 differs from that in Section A.4 *only* by its structure. The first decomposition makes no attempt to isolate the representations of the task modules. For example, `*line-storage*` is accessed directly by the circular shifter and the alphabetizer, exposing the fact that it is a list. The data decomposition, conversely, isolates the representation inside line access routines. Thus it is expected that the latter will be easier to modify than the former; otherwise they are the same, using the same representations and algorithms.

The following demonstrates that it is possible to apply meaning-preserving transformations using the restructuring tool to migrate from the functional modularization to the data modularization. By first restructuring with the tool and then performing functional changes, the tool guarantees that the meaning of the program is preserved and performs the necessary non-local changes to preserve meaning. Thereafter, the likely changes—those to representation, for example—are localized within a module. So although the functional changes are not themselves automated, they are local, and thus easier to reason about. This benefit is demonstrated with an example.

As in the restructuring of the matrix multiply (Chapter 2.6), the tool commands shown below are slight variations of the actual commands used in the tool to make them more readable. Especially, <angle braces> are used to give names to a location in the program being restructured. See Chapter 2 for details on the tool's user interface and the transformations the tool supports.

A.1 The Restructuring

The most evident structural property of the functional decomposition is that almost every module directly accesses the data of other modules, such as `*line-storage*` of the input module. Also evident is that several functions are declared local to a module function and need to be moved out to make them reusable across modules. In other cases, sequences of in-line code must be abstracted as real functions before they can be reused. The primary goal, then, is to create the appropriate abstractions to isolate all references to exposed data. In the process, the module structure will evolve to the data decomposition described by Parnas.

Starting with the initial functional decomposition in Section A.3, it is apparent that the input module's direct reference to the `*line-storage*` must be removed so that line-storage can begin to stand on its own. The first step is to move out local function `insline`, which will become a function of the new line-storage module:

```
(move-expr <insline definition> :before <putfile definition>)
(ungroup <empty letrec>)  ; remove empty letrec, too
```

The result, then, is this change:

```
(define insline
 (lambda (line)
   (if line (setq *line-storage* (cons line *line-storage*)) nil)))

(define putfile
 (lambda (linelist)
   (do ((restlist linelist (cdr restlist)))
       ((null? restlist)
        nil)
     (insline (car restlist)))))                ; fixed
```

Note that the input module (function `putfile`) no longer references `*line-storage*` directly, but instead calls `insline`, insuring that any change to the representation of the line storage is now isolated from the input module.

Now consider the function `cssetup`, which contains several exposed references to `*line-storage*`. One is the binding declaration

```
(numlines (length *line-storage*))
```

for iterating through the lines of the store, and another is

```
(set! numwords (length (list-ref *line-storage* lineno)))
```

used to iterate through the circular shifts of an individual line. These are extracted and placed next to `insline` to become part of the line-storage module:

```
(extract-function <(length *line-storage*)> 'lines
 :after <insline definition>)

(extract-function <(length (list-ref *line-storage* lineno))> 'words
 :old-new-name-pairs (list (list <reference to lineno>))
 :after <line definition>)
```

So now the line-storage module consists of the original representation declaration, plus these functions:

```
(define *line-storage* nil)

(define insline
 (lambda (line)
   (if line (setq *line-storage* (cons line *line-storage*)) nil)))

(define lines (lambda ()
                (length *line-storage*)))

(define words (lambda (lineno)
                (length (list-ref *list-storage* lineno))))
```

and `cssetup` now looks like:

```
(define cssetup (lambda ()
  (letrec
   ((allwords (lambda (ls)
      (do ((restls ls (cdr restls))
           (sum 0 sum))
          ((null? restls) sum)
        (set! sum (+ sum (length (car restls))))))))

     (let ((numcslines (allwords *line-storage*))    ; remaining
           (cslineno 0)
           (numlines (lines))                        ; fixed
           (numwords nil))
       (set! *circ-index* (make-vector numcslines))
       (do ((lineno 0 (1+ lineno)))
           ((= lineno numlines)
            nil)
         (set! numwords (words lineno))              ; fixed
         (do ((wordno 0 (1+ wordno)))
             ((= wordno numwords)
              nil)
           (vector-set! *circ-index* cslineno (list lineno wordno))
           (set! cslineno (1+ cslineno))))))))
```

Unfortunately, there is still one reference to `*line-storage*` remaining in the call to `allwords`. This will be fixed later on.

Now attention is turned to the circular shift module. First, the definition of the comparison function for sorting shifted lines, `csline<=`, is in the alphabetizer, rather than in the shift module. This function contains information about how shifted lines (and lines for that matter!) are represented. It also is hiding the functions `csword` and `cswords`, which are quite reusable functions. The following tool commands moves `csline<=` to its rightful place, and exposes `csword` and `cswords`.

```
(move-expr <csline<= definition> :after <*circ-index* definition>)
(move-expr <csword definition> :after <*circ-index* definition>)
(move-expr <cswords definition> :after <*circ-index* definition>)
(ungroup <empty letrec>)  ; remove unneeded letrec
```

These functions still directly access the line store, which will be addressed later, along with the remaining reference in `cssetup`. However, the function `alph` is now free of the details of how lines are compared, although it still directly accesses the representation of the circular shifter, `*circ-index*`, in fact in an analogous fashion to how the shifter accessed the line store:

```
(let ((numitems (length *circ-index*)))
```

for retrieving the number of shifts, needed for allocated the alphabetization store, and

```
(vector-set! *alph-index* i (vector-ref *circ-index* i)))
```

for initializing the alphabetization store with the unsorted elements. (Fortunately, the representation of shifts—a list containing the line number and a word number within the line—is no longer exposed in the alphabetizer, because it is isolated within `csline<=`.) These two violations of representation hiding are removed with two function extractions:

```
(extract-function <(length *circ-index*)> 'cslines
 :after <*circ-index* definition>)

(extract-function <(vector-ref *circ-index* i)> 'csline
 :old-new-name-pairs (list (list <i reference>))
 :after <*circ-index* definition>)
```

Although there is more restructuring to be done on it, the circular shift module begins to shape up like this:

```
(define *circ-index* nil)

(define cswords (lambda (shift ls)
                  (length (list-ref ls (car shift)))))

(define csword
 (lambda (shift wordno ls)
   (let* ((lno (car shift))
          (fwno (cadr shift)))
     (list-ref (list-ref ls lno)
      (modulo (+ fwno wordno) (length (list-ref ls lno)))))))

(define csline<=
 (lambda (shift1 shift2 ls)
   (let ((lasti
          (min (cswords shift1 ls) (cswords shift2 ls)))
         (result nil)
         (done? nil))
     (do ((i 0 (1+ i)))
         (done?
          result)
       (let ((maxed? (= i lasti))
             (cword1 (symbol->string (csword shift1 i ls)))
             (cword2 (symbol->string (csword shift2 i ls))))
         (if (or maxed? (not (string=? cword1 cword2)))
             (begin (set! done? t)
              (set! result
               (if maxed? (<= lasti (cswords shift2 ls))
                   (string<=? cword1 cword2))))
             nil))))))

(define cssetup
 (lambda ()
   (letrec
    ((allwords
      (lambda (ls)
        (do ((restls ls (cdr restls))
             (sum 0 sum))
            ((null? restls)
             sum)
          (set! sum (+ sum (length (car restls))))))))
    (let ((numcslines (allwords *line-storage*))
          (cslineno 0)
          (numlines (lines))
          (numwords nil))
        (set! *circ-index* (make-vector numcslines))
        (do ((lineno 0 (1+ lineno)))
            ((= lineno numlines)
             nil)
          (set! numwords (words lineno))
          (do ((wordno 0 (1+ wordno)))
              ((= wordno numwords)
               nil)
            (vector-set! *circ-index* cslineno (list lineno wordno))
            (set! cslineno (1+ cslineno)))))))
```

Turning attention now to the output module, which for this simple implementation just prints out the sorted shift lines to the terminal, several violations are evident: the function `csline` local to `allalphcslines` contains information about the representation of shifted lines (so it can print them), `allalphcslines` directly accesses the `*alph-index*` representation in two locations, and `*line-storage*` is directly accessed as well.

First the local function `csline` must be moved to the shift module, but this requires renaming it first, since there is already a function by that name.

```
(rename-variable
 <allalphcsline's local csline definition> 'printable-csline)

(move-expr <printable-csline definition> :after <csline definition>)
(ungroup <empty letrec>)  ; remove useless letrec
```

Next the access violations to `*alph-index*` must be removed: one retrieving the size of `*alph-index*` the other retrieving elements of `*alph-index*`.

```
(extract-function <(vector-ref *alph-index* i)> 'alphcsline
 :old-new-name-pairs (list (list <i reference>))
 :after <*alph-index* definition>)

(extract-function <(length *alph-index*)> 'alphcslines
 :after <*alph-index* definition>)
```

After these extractions, `allalphcslines` looks like this:

```
(define allalphcslines
 (lambda ()
      (let ((numcslines (alphcslines)))
        (do ((i 0 (1+ i)))
            ((= i numcslines)
            nil)
          (printf "~s~%" (printable-csline (alphcsline i) *line-storage*))))))
```

Now there is one remaining problem throughout the program, the direct references to `*line-storage*`. These permeate every module, rather than just the circular shift module, because a shift is represented as a list containing a line number and a word index, meaning that to actually access the shifted line for comparison or printing requires accessing the line storage. To overcome these violations requires two steps. First, line storage is explicitly passed to many functions, and to abstract the references to the storage away requires inlining these parameters. Note that all of the function definitions with these parameters are isolated in the shift module, although the functions are called in other modules. Since the parameter is passed through more than one level, this must be done repeatedly. Second, once the references are inlined they need to be abstracted as functions of the line storage module. First the inlinings:

```
(binding-to-expr <ls parameter in csline<= definition>)
(binding-to-expr <ls parameter in cswords>)
(binding-to-expr <ls parameter in csword>)
(binding-to-expr <ls parameter in printable-csline>)
(binding-to-expr <ls parameter in allwords>)
```

For each to succeed requires that all the calls of the function take the same expression as the argument to `ls`. This is because each call represents an independent binding of the parameter `ls`, but when inlined it must be represented by just one parameter. Of course, in these instances `ls` is a reference to `*line-storage*`. The result of inlining the parameters is that `*line-storage*` is now explicitly referenced in all of these functions, except `csline<=`, in which there is no remnant of it because its uses were transitively inlined:

```
(define printable-csline
 (lambda (shift)
   (let* ((lno (car shift))
          (fwno (cadr shift))
          (wrdcnt (length (list-ref *line-storage* lno)))
          (revcs nil))
     (do ((i 0 (1+ i)))
         ((= i wrdcnt)
          nil)
       (set! revcs
        (cons (list-ref (list-ref *list-storage* lno)
              (modulo (+ i fwno) wrdcnt)) revcs)))
     (reverse revcs))))

(define cswords (lambda (shift)
                  (length (list-ref *line-storage* (car shift)))))

(define csword
 (lambda (shift wordno)
   (let* ((lno (car shift))
          (fwno (cadr shift)))
     (list-ref (list-ref *line-storage* lno)
      (modulo (+ fwno wordno) (length (list-ref *line-storage* lno)))))))

(define csline<=
 (lambda (shift1 shift2)
   (let ((lasti (min (cswords shift1) (cswords shift2)))
         (result nil)
         (done? nil))
     (do ((i 0 (1+ i)))
         (done?
          result)
       (let ((maxed? (= i lasti))
             (cword1 (symbol->string (csword shift1 i)))
             (cword2 (symbol->string (csword shift2 i))))
         (if (or maxed? (not (string=? cword1 cword2)))
             (begin (set! done? t)
              (set! result
               (if maxed? (<= lasti (cswords shift2))
                   (string<=? cword1 cword2))))
             nil))))))

(define cssetup
 (lambda ()
   (letrec
    ((allwords
      (lambda ()
        (do ((restls *line-storage* (cdr restls))
             (sum 0 sum))
             ((null? restls)
              sum)
           (set! sum (+ sum (length (car restls)))))))
     (let ((numcslines (allwords))
           (cslineno 0)
           (numlines (lines))
           (numwords nil))
       (set! *circ-index* (make-vector numcslines))
       (do ((lineno 0 (1+ lineno)))
           ((= lineno numlines)
            nil)
         (set! numwords (words lineno))
         (do ((wordno 0 (1+ wordno)))
             ((= wordno numwords)
              nil)
           (vector-set! *circ-index* cslineno (list lineno wordno))
           (set! cslineno (1+ cslineno)))))))
```

To take care of the access violations in `printable-csline`, `csword` and `cswords`, the accesses to `*line-storage*` must now be abstracted. The expression `(length (list-ref *line-storage* (car shift)))` can be represented as the `words` function, already extracted earlier. Likewise, the expression `(list-ref *line-storage* lno)` in `csword` can be extracted as the `line` function, with a parameter for `lno` of the line-storage module. These are accomplished with the following three commands:

```
(scope-sub-call <words definition>)

(scope-extract-function <(list-ref *line-storage* lno)> 'line
 :old-new-name-pairs (list (list <lno reference> 'lineno))
 :after <*line-storage* definition>)

(scope-sub-call <line definition>)
```

The transformation `scope-sub-call` finds all sequences of expressions in the program that represent the same function as the one passed as the argument, and replaces the matched expressions with a call to the function. The first transformation finds two expressions matching `words`, the body of `cswords`, but the new `line` function matches in a few of locations, including in the `words` function. The match in `words` could have been rejected reasonably by the engineer, because the `words` definition is in the line-storage module, and so its direct access of `*line-storage*` was not an access violation, albeit redundant. The result is:

```
(define line (lambda (lineno)
               (list-ref *line-storage* lineno)))
   . . .

(define words (lambda (lineno)
                (length (line lineno))))
   . . .

(define printable-csline
 (lambda (shift)
   (let* ((lno (car shift))
          (fwno (cadr shift))
          (wrdcnt (words lno))
          (revcs nil))
     (do ((i 0 (1+ i)))
         ((= i wrdcnt)
          nil)
       (set! revcs
        (cons (list-ref (line lno) (modulo (+ i fwno) wrdcnt)) revcs)))
     (reverse revcs))))

(define cswords (lambda (shift)
                  (words (car shift))))

(define csword
 (lambda (shift wordno)
   (let* ((lno (car shift))
          (fwno (cadr shift)))
     (list-ref (line lno) (modulo (+ fwno wordno) (length (line lno)))))))
```

At this point there is only one remaining access violation according to Parnas's preferred data decomposition, the access of `*line-storage*` in function `allwords`, located inside `cssetup`. In fact, the function itself belongs in the line-storage module, since it is iterating through the storage, so it is moved out of `cssetup`:

```
(move-expr <allwords definition> :after <words definition>)
(ungroup <empty letrec>)  ; remove empty letrec
```

The final result is presented in Section A.4, *sans* comments. It is not quite done, actually. Although unspecified for this example, the representation of a line has not been isolated. For example, functions `csword` and `printable-csline` use `list-ref` to access the words of a retrieved line. Changing the representation to vectors would require changes in at least two modules.

A.2 Evaluation

The restructuring tool has successfully aided the restructuring the task-oriented implementation of KWIC to a representation-oriented structure. All the non-local changes necessary for preserving consistency of the meaning of the program were made by the tool, but the choice of the structure was under the control of the engineer. Parnas has already discussed the benefits of the resulting structure [Parnas 72], but consider the following example.

Suppose it is desired to no longer store circular shifts because they are large in number compared to the number of regular lines. This is easy to implement because, fascinatingly enough, the i^{th} circularly shifted line starts with the i^{th} word in the line-storage. Hence, to compute the i^{th} shift merely requires finding the line with the i^{th} word in the line-storage and creating a line shifted to start with that word. Although it is not possible (using the existing tool, anyway) to restructure from the existing data decomposition to this new implementation of circular shifts, the necessary changes are confined to the circular shift module. On the other hand, in the original functional decomposition the representation of shifts was dispersed through the shifter, alphabetizer and output modules. Hence the restructuring has successfully localized a property that was dispersed in the original. The new version should cost less to modify.

A.3 Original Control Decomposition of KWIC

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; INPUT MODULE
;;;
```

```
;;;
;;; list of list of words.
;;;
(define *line-storage* nil)

(define putfile (lambda (linelist)
  (letrec
    ;;
    ;; Adds a line.  By convention, lineno=0 implies the first line.
    ;;  We assume the input is a list of symbols.  We convert to strings
    ;;  for comparisons and such.
    ;;
    ((insline (lambda (line)
       (if line (set! *line-storage* (cons line *line-storage*))))))

  (do ((restlist linelist (cdr restlist)))
      ((null? restlist) nil)
    (insline (car restlist))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; CIRCULAR SHIFTER
;;;
;;;  This module creates the illusion (or reality) that the lineholder
;;;   has had all the circular shifts of lines inserted for all lines.
;;;   For line i < j, all of i's shifts come before the j's.  The shifts
;;;   are inserted in order starting from the original line.
;;;
;;;  Amazing fact:  for a line with N words, there are N circular shifts.
;;;  This means that shift M is the line containing the Mth word in the file,
;;;  with the first word of the shift being the Mth word.
;;;


(define *circ-index* nil)

;;;
;;; Build an index of circulars.  These are represented as pairs of
;;;  (lineno, wordno).
;;;
(define cssetup (lambda ()
  (letrec
   ((allwords (lambda (ls)
      (do ((restls ls (cdr restls))
           (sum 0 sum))
```

```
          ((null? restls) sum)
        (set! sum (+ sum (length (car restls)))))))

  (let ((numcslines (allwords *line-storage*))
        (cslineno 0)
        (numlines (length *line-storage*))
        (numwords nil))
    (set! *circ-index* (make-vector numcslines))
    (do ((lineno 0 (1+ lineno)))
        ((= lineno numlines) nil)
      (set! numwords (length (list-ref *line-storage* lineno)))
      (do ((wordno 0 (1+ wordno)))
          ((= wordno numwords) nil)
        (vector-set! *circ-index* cslineno (list lineno wordno))
        (set! cslineno (1+ cslineno)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; ALPHABETIZING MODULE
;;;
;;;  This contains function alph.  It creates an array just like CS's but
;;;   is sorted.
;;;

(define *alph-index* nil)

(define alph (lambda ()
  (letrec
    ;; Says if shiftno1 is less than or equal to shiftno2
   ((csline<= (lambda (shift1 shift2 ls)
     (letrec
       ;; Return the word on line number shiftno, at word number wordno in the
       ;;  line.  Result is a string.
      ((csword (lambda (shift wordno ls)
         (let* ((lno  (car shift))
                (fwno  (cadr shift))) ; number of the first word in the shift
           (list-ref (list-ref ls lno)
                     (modulo (+ fwno wordno) (length (list-ref ls lno)))))))

       ;; Returns the number of words in line number shiftno
       (cswords (lambda (shift ls)
           (length (list-ref ls (car shift))))))

      (let ((lasti (min (cswords shift1 ls)
                        (cswords shift2 ls)))
        (result nil)
        (done? nil))
    (do ((i 0 (1+ i)))
        (done? result)
      (let ((maxed? (= i lasti))
            (cword1 (symbol->string (csword shift1 i ls)))
            (cword2 (symbol->string (csword shift2 i ls))))
        (if (or maxed? (not (string=? cword1 cword2)))
            (begin
             (set! done? t)
             (set! result
                   (if maxed?
                       (<= lasti (cswords shift2 ls))
                       (string<=? cword1 cword2))))))))))

;; Swap the values at two indices in the vector
(swap-indices (lambda (vec i j)
  (let ((temp (vector-ref vec i)))
    (vector-set! vec i (vector-ref vec j))
    (vector-set! vec j temp))))

;; Look at each cs-line from start to end and put its index in the upper or
;;  lower half of *alph-index*.  An equal comparison defaults to the left
;;  side.
(qsplit (lambda (start end split)
  (let ((low (1+ start))  ; start one below bot, and use bot as <= split
        (high end))

    ;; swap the split and start so split doesn't get mixed in swaps.
    (swap-indices *alph-index* start split)
    (set! split start)

    ;; do split
    (do ()
        ((> low high) nil)
      (if (csline<= (vector-ref *alph-index* low)
                    (vector-ref *alph-index* split)
                    *line-storage*)
          (set! low (1+ low))
          (begin
           (swap-indices *alph-index* low high)
           (set! high (1- high)))))

    ;; On exit of loop, we are guaranteed that (1- low) is in the low end.
    ;;  In the worst case, it is start (i.e., split).  So we swap this
    ;;  with the split, and then qalph will sort everything above and
```

```
        ;;  below (1- low).
        (swap-indices *alph-index* split (1- low))

        (1- low))))

    ;; Quicksort the shifted lines from start to end.
    (qalph (lambda (start end)
      (if (< start end)
          (let* ((split start)
                 (middle (qsplit start end split)))
            (begin
             (qalph start (1- middle))
             (qalph (1+ middle) end)))))))

  ;; THE REAL CODE
  (let ((numitems (length *circ-index*)))
    (set! *alph-index* (make-vector numitems))
    (do ((i 0 (1+ i)))
        ((= i numitems) nil)
      (vector-set! *alph-index* i (vector-ref *circ-index* i)))
    (qalph 0 (1- numitems)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; OUTPUT MODULE
;;;



(define allalphcslines (lambda ()
 (letrec
   ;; builds a circularly shifted line storage and a shiftspec
  ((csline (lambda (shift ls)
     (let* ((lno    (car shift))
            (fwno   (cadr shift))
            (wrdcnt (length (list-ref ls lno)))
            (revcs nil))
     (do ((i 0 (1+ i)))
         ((= i wrdcnt))
       (set! revcs
             (cons (list-ref (list-ref ls lno) (modulo (+ i fwno) wrdcnt))
                   revcs)))
     (reverse revcs)))))

  (let ((numcslines (length *alph-index*)))
    (do ((i 0 (1+ i)))
        ((= i numcslines) nil)
      (printf "~s~%" (csline (vector-ref *alph-index* i) *line-storage*))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; MASTER CONTROL
;;;

(putfile (list '(a b c d) '(one) '(hey this is different) '(a b c d)))
(cssetup)
(alph)
(allalphcslines)
```

A.4 Restructured Data Decomposition of KWIC

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; LINE-STORAGE MODULE
;;;

(define *line-storage* nil)

(define line (lambda (lineno)
               (list-ref *line-storage* lineno)))

(define lines (lambda ()
                (length *line-storage*)))

(define words (lambda (lineno)
                (length (line lineno))))

(define allwords
 (lambda ()
   (do ((restls *line-storage* (cdr restls))
        (sum 0 sum))
       ((null? restls)
        sum)
     (set! sum (+ sum (length (car restls)))))))

(define insline
 (lambda (line)
   (if line (set! *line-storage* (cons line *line-storage*)) nil)))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; INPUT MODULE
;;;


(define putfile
 (lambda (linelist)
   (do ((restlist linelist (cdr restlist)))
       ((null? restlist)
        nil)
     (insline (car restlist)))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; CIRCULAR SHIFTER
;;;

(define *circ-index* nil)

(define cslines (lambda ()
                  (length *circ-index*)))

(define csline (lambda (i)
                 (vector-ref *circ-index* i)))

(define printable-csline
 (lambda (shift)
   (let* ((lno (car shift))
          (fwno (cadr shift))
          (wrdcnt (words lno))
          (revcs nil))
     (do ((i 0 (1+ i)))
         ((= i wrdcnt)
          nil)
       (set! revcs
        (cons (list-ref (line lno) (modulo (+ i fwno) wrdcnt)) revcs)))
     (reverse revcs))))

(define cswords (lambda (shift)
                  (words (car shift))))
```

```
(define csword
 (lambda (shift wordno)
   (let* ((lno (car shift))
          (fwno (cadr shift)))
     (list-ref (line lno) (modulo (+ fwno wordno) (length (line lno)))))))

(define csline<=
 (lambda (shift1 shift2)
   (let ((lasti (min (cswords shift1) (cswords shift2)))
         (result nil)
         (done? nil))
     (do ((i 0 (1+ i)))
         (done?
          result)
       (let ((maxed? (= i lasti))
             (cword1 (symbol->string (csword shift1 i)))
             (cword2 (symbol->string (csword shift2 i))))
         (if (or maxed? (not (string=? cword1 cword2)))
             (begin (set! done? t)
              (set! result
               (if maxed? (<= lasti (cswords shift2))
           (string<=? cword1 cword2))))
             nil))))))

(define cssetup
 (lambda ()
    (let ((numcslines (allwords))
          (cslineno 0)
          (numlines (lines))
          (numwords nil))
      (set! *circ-index* (make-vector numcslines))
      (do ((lineno 0 (1+ lineno)))
          ((= lineno numlines)
           nil)
        (set! numwords (words lineno))
        (do ((wordno 0 (1+ wordno)))
            ((= wordno numwords)
             nil)
          (vector-set! *circ-index* cslineno (list lineno wordno))
          (set! cslineno (1+ cslineno)))))))


;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; ALPHABETIZING MODULE
;;;
```

```
(define *alph-index* nil)

(define alph
 (lambda ()
   (letrec
    ((swap-indices
      (lambda (vec i j)
        (let ((temp (vector-ref vec i)))
          (vector-set! vec i (vector-ref vec j))
          (vector-set! vec j temp))))
     (qsplit
      (lambda (start end split)
        (let ((low (1+ start))
              (high end))
          (swap-indices *alph-index* start split)
          (set! split start)
          (do ()
              ((> low high) nil)
            (if
             (csline<= (vector-ref *alph-index* low)
              (vector-ref *alph-index* split))
             (set! low (1+ low))
             (begin
      (swap-indices *alph-index* low high)
      (set! high (1- high)))))
          (swap-indices *alph-index* split (1- low))
          (1- low))))
     (qalph
      (lambda (start end)
        (if (< start end)
            (let* ((split start)
                   (middle (qsplit start end split)))
              (begin (qalph start (1- middle)) (qalph (1+ middle) end)))
            nil))))
    (let ((numitems (cslines)))
      (set! *alph-index* (make-vector numitems))
      (do ((i 0 (1+ i)))
          ((= i numitems)
           nil)
        (vector-set! *alph-index* i (csline i)))
      (qalph 0 (1- numitems))))))


(define alphcsline (lambda (i)
                     (vector-ref *alph-index* i)))
```

```
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
;;;
;;; OUTPUT MODULE
;;;

(define allalphcslines
 (lambda ()
    (let ((numcslines (cswords)))
      (do ((i 0 (1+ i)))
          ((= i numcslines)
           nil)
        (printf "~s~%" (printable-csline (alphcsline i)))))))
```

Vitae

William G. Griswold was born in Red Bank, New Jersey, on November 13, 1962, growing up in Little Silver, and later, Middletown, New Jersey. Bill graduated from Middletown High School North in 1981. He received his Bachelor of Arts (Highest Honors) from the University of Arizona, in Tucson, in 1985 and Master of Science from the University of Washington, Seattle, in 1988. He completed his doctoral studies at the University of Washington in 1991.