THE REPLAY OF PROGRAM DERIVATIONS

BY

ROBERT W. HASKER

B.S., Wheaton College, 1984

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1995

Urbana, Illinois

THE REPLAY OF PROGRAM DERIVATIONS

Robert W. Hasker, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1995
Uday S. Reddy, Advisor

A promising though radical approach to software development is to write *formal specifications* and then *derive* implementations by applying sequences of formal steps. This is often known as *transformational implementation.* An advantage of this approach is increased consistency between specifications and implementations. But perhaps a more important advantage is the potential for maintaining specifications rather than implementations. Because derivation sequences formally describe how implementations are constructed, the user can modify specifications and then *replay* the derivations to obtain new implementations.

This thesis discusses replaying derivations in an interactive environment. We identify several problems which need to be addressed to make replay both autonomous and robust. We present our implemented replay system, ReFocus, and describe how it addresses these problems. In particular, we present methods for updating derivations to match new specifications and for verifying the acceptability of implementations produced by replay.

Updating derivations represents the technical core of this thesis. The system constructs analogical maps between specifications and applies them to derivations to solve new problems. Traditionally, analogical maps are constructed using *first-order generalization*, or *anti-unification.* However, we show that first-order generalization is too inflexible for program derivations. In its place, we propose using *second-order generalization.* We give a definition of second-order generalization, identify classes of terms for which such generalizations exist, and show that it is computable. We then identify an important subset of generalizations and give (under assumptions) a quadratic-time algorithm for computing this subset. Finally, we show that these generalizations provide the necessary flexibility for replay.

We give a number of examples illustrating the usefulness of replay on small to medium-sized problems. For each, we show that ReFocus is able to obtain useful results. This demonstrates that replaying program derivations is both feasible and useful.

To my wife, Jill, and my parents, Nancy and William.

Acknowledgements

First, I wish to thank my advisor, Uday Reddy, for his considerable help while working on this thesis, including suggesting interesting research directions, helping improve my technical writing skills, and teaching me much about research. I also wish to thank the members of my committee for their help in a number of places: John Gray, for discussing some of the theoretical foundations of generalization; Mehdi Harandi, for suggesting replay issues to consider; and Sam Kamin and Nachum Dershowitz, for suggesting significant improvements to the presentation.

I also wish to thank my fellow students for providing friendship and support over the years. In particular, I wish to thank Francois Bronsard, Bill Harrison, and T. K. Lakshman for reviewing early drafts of this work and making helpful suggestions. I also thank Francois for patiently explaining many formal aspects of program derivation, and Bill for drawing my attention to a number of interesting replay examples. Without their help and encouragement, this thesis would have taken significantly longer to complete.

Finally, I wish to thank my family: my parents Nancy and Bill, for always believing in me; my brother Kevin, for having confidence that I would finish; my cat Hazel, for providing much-needed entertainment in the final stages; and most of all, my wife Jill, who has endured more than I dared ask in helping me to complete this work. Without their love and support, I would never have finished.

This research was supported in part by a grant from the Motorola Corporation. I am thankful for their support.

Table of Contents

Chapter 1

Introduction

The usefulness of automating the software development process has not always been generally recognized. In the early years of computing, Don Gillies was given the job of implementing some algorithms on John von Neumann's machine. To help, Gillies wrote what would be considered today to be a primitive assembler. Von Neumann did not approve, claiming that "it is a waste of a valuable scientific computing instrument to use it to do such clerical work" [Jon92]. Today, however, hardware is much cheaper, especially relative to the costs of producing software, so the usefulness of automating more of the software development process is much clearer. A popular—and worthwhile—approach is to provide more tools within the current paradigm. But the amount of automation which can be supported by the current paradigm appears to be limited. We believe that to benefit fully from automation, a new software development paradigm is needed.

The paradigm advocated in this thesis is based on specifying a system formally and using formal methods to *derive* an implementation from the specification. This radical approach to software development, suggested by many but perhaps best described in [BCG83, SS83], allows increased automation in many ways. One of the more important of these is in system maintenance. System maintenance is one of the most expensive phases of any large project, so increasing the amount of automated support for maintenance should have a large payoff. In this new paradigm, instead of modifying the implementation directly, the user can modify the specification and reimplement it by *replaying* the derivation.

Replaying program derivations raises a number of issues. This thesis addresses three of the more important ones. First, what characteristics improve the replayability of derivations. Second, how to know when replay has been successful. Third, how to update a derivation so that it applies to a new or modified problem. To address this last problem, we present a general definition of analogy based on higher-order generalization and show how to use such analogies in replay.

1.1 The Transformational Implementation Paradigm

As in [BCG83], we believe the current software development paradigm (typified by the waterfall model) has a number of problems. One problem is that the paradigm relies on using informal methods to convert (usually informal) specifications to code. This makes it difficult to verify that the two are consistent. A common, partial solution is to formalize the specification by building a prototype. But the primary purpose of a prototype is to communicate with the customer; the final implementation usually has little relationship to it except for satisfying similar requirements.[1] Another problem with the paradigm is that the design documentation is incomplete. It does not record all of the decisions made by the developer, only those which the developer finds "interesting." There is no assurance that this information will be useful to the next person to work on the system.

However, the most important problem with the current paradigm is that maintenance is done on the *product* of the process, the implementation, rather than its *source*, the specification. That is, the primary result of maintenance is a change in code. Implementation distributes information by replacing relatively simple abstractions by complex data representations, algorithms, and interactions. The complex interactions are often not well understood, so making changes to one part of a program can easily introduce errors in other parts. Thus maintaining the implementation means that maintenance is directed at the part of the system that is hardest to understand.

Another drawback of maintaining implementations is that it makes the other problems in the paradigm more acute. There is no formal link between the specification and implementation, so the methodology relies on the maintainer to update the specification to match the new implementation. Even if the maintainer understands the specification well enough to update it correctly, updates may not be made due to pressure to move on to the next job. Likewise, maintaining the prototype along with the implementation is expensive, so the prototype is usually discarded after the initial development. Finally, changes weaken the structure of the system. This makes it harder to verify the correctness of the new system and harder to make changes in the future.

To address these problems, a number of researchers, including [BGW76, BCG83, Dar81, Fea82, Knu74, Red90a, SS83, Wil83], suggest using semi-automated tools to convert formal specifications into implementations. This paradigm, called both *transformational implementation* and *transformational derivation*, relies on equivalence-preserving rules to provide a formal link between the specification and implementation. The user starts from a formal, executable specification and then chooses an appropriate sequence of steps to transform the specification

[1] Unless the prototype becomes the final implementation!

into an efficient program. We call this sequence of steps a *derivation.* Because each step preserves equivalence, the program and its correctness proof are developed simultaneously.

This paradigm has the potential to improve the development process in a number of ways. The relationship between specifications and implementations is formalized, so the number of clerical errors is reduced. The paradigm supports proving properties about the specification and its implementation; this can increase the user's confidence in the resulting program. Finally, this paradigm allows maintenance to be done on specifications. The sequence of steps from specifications to implementations is completely described, so it can be a guide for propagating changes from the first to the second.

This method is an improvement because it would appear that specifications are easier to maintain; they do not need to be efficient and so can be written for clarity. Modifying specifications is not trivial, but at least the information is usually more localized and explicit. A second benefit of maintaining specifications is that since the specification is executable, it serves as a prototype which remains consistent with the implemented system and can provide a convenient basis for testing proposed changes. But the most important benefit is that fewer bugs are likely to be introduced (or reintroduced) during maintenance. If a change makes a previous optimization impossible, the system can detect the inconsistency and warn the developer of the problem. Since maintenance is usually the costliest part of a project, improving the support for maintenance may prove to be the most important benefit of the transformational implementation paradigm.

1.2 Change Propagation by Replay

There are a number of significant practical and theoretical issues in applying formal methods develop programs. Theoretical issues include what sort of consistency claims can be made for mechanical proofs: in mathematics, proofs are accepted only after being examined by the community. It seems that the social process plays an important role in verifying a result [DMLP79]. Even if a proof system correctly implements an accepted methodology, random hardware errors may cause the system to generate incorrect proofs during any particular run. [SS83] argues that in spite of these issues, useful transformation systems can be built and relied upon in the same way that we rely on compilers to translate high-level languages into machine code. Practical issues include the difficulty of writing formal specifications and the difficulty of constructing derivations and proofs. Progress has been made on applying formal methods to medium-scale systems, but as illustrated in [Coh88], it is still very time-consuming to obtain useful results. In any case, it is impossible to mechanically validate very large, production-quality systems.

This thesis addresses a practical issue: how to propagate changes from specifications to implementations. It is very likely that proving large systems meet their specifications will always be expensive. But if formal methods make it easier to maintain them, then it may be possible to recover the initial expense during the later phases of a project. Thus a demonstration of the feasibility of automatically propagating changes helps motivate research in applying formal methods to large systems.

One way to propagate changes is to build a set of transforms which modify the specification and implementation simultaneously. However, this method has the same problem as informal methods: it is difficult to ensure that the changes preserve consistency. A refinement is to compare the original and new specifications, transform the program accordingly, and then use a theorem prover to either show the new program is correct or to make the appropriate fixes. But a limitation of this method is that for large problems, it is difficult to construct a correct program matching the specification.

The transformational implementation paradigm suggests an alternative method. As suggested in [BCG83, Dar81, Fic85, Red90a, SS83, Wil83], the system can record the steps used to derive the implementations and reexecute, or *replay* them, after modifying specifications. This shifts the burden of maintaining consistency to the underlying transformational system. Another advantage is that the derivation history, as part of the design documentation, is automatically maintained along with the code. This improves the quality of the documentation by insuring that the derivation describes the current implementation. Thus replay can be used to increase the amount of automation in maintenance.

As others, we have found replay to be also useful for transferring design knowledge between problems in different domains. We call this *derivation-by-analogy.*[2] Since deriving similar programs often involves similar steps, replaying derivations can save the user time by reusing the process of constructing implementations. It also makes it easier to experiment with different ways of optimizing a specification, potentially improving the quality of the implementation by allowing the user to explore ideas for which there may not otherwise be time. In addition to automating maintenance, this thesis addresses using replay for derivation-by-analogy.

Designing a useful replay mechanism poses interesting challenges because reexecuting derivations can fail for several reasons. First, replay may be unable to satisfy the preconditions of a step. Second, it may be unable to correctly update a reference to the specification that is stored in a step. Third, even if a step is applied, it may not achieve the original purpose and the new program may be too inefficient. To be useful, replay must be *robust*; it must be able to reexecute derivations in spite of changes in the specifications and supporting theory.

[2]As opposed to *derivational analogy* [Car86], which is a specific form of reuse as discussed in Section 1.3.

This research address two primary issues of making replay robust. First, replay must be able to *propagate differences between specifications*: it must determine the changes made to a specification and apply them to the derivation so that it is specialized to the new problem. Without this, replay is overly sensitive to changes. Secondly, replay must *test for acceptability*: it must ensure that the newly derived program is as efficient as the original. This allows the user to concentrate on new work instead of checking and rechecking the results of replaying derivations.

While replay must be robust, it must also be *autonomous*. If the user spends too much time providing background information and making decisions, then it becomes easier to reconstruct the derivation by hand. Thus in systems where there is no extensive domain knowledge, replay must make a "best guess" based on domain-independent heuristics.

A major part of our solution to these problems, propagating differences and testing for acceptability, is based on building an analogy between the old and new specifications. Since known methods for building analogies are too inflexible or too expensive for replay, this research gives a new definition of analogy based on higher-order generalization. This type of analogy provides a basis for making decisions based solely on term structure.

While the focus of this thesis is on replaying program derivations, replay is also useful in semi-automated theorem proving. The basic problem in both program derivation and theorem proving is the same: the need to control search. In both, either the system or the user must decide what steps to apply and how to apply them. In both, replay can be used to capture information about search control and apply it to new problems. The difference is in the product. In theorem proving, the goal is exhibit a proof. The only constraint on the proof is that all inferences are valid. In program derivation, the goal is to exhibit a program satisfying performance requirements. Thus program derivation is constrained by what sort of program can be produced. This makes program derivation, and replaying program derivations, a harder problem. However, since program derivation is a form of theorem proving, much of the discussion of replay in this thesis also applies to theorem proving. We use the term "proof development" to refer to both theorem proving and program derivation.

1.3 Reuse and Related Work

Replaying proof developments is a specific form of *reuse*. Given a problem and its solution, the intuition is that the solution should be reusable when confronted with a new, but similar problem. Reuse can take many forms:

- reusing components when problems are identical,
- reusing solutions when problems are equivalent,

- reusing solutions when problems are analogous, and
- reusing the process of constructing solutions.

Reusing components is the goal of much research on programming languages. For instance, object-oriented programming supports reuse by identifying building blocks that can be lifted from one problem and incorporated into others [GR83, Str91]. Another area of research, parametric polymorphism, supports reuse by abstracting common procedures so they can be applied to different sorts of objects [Mil78, CW85]. In general, research into creating reusable components focuses on introducing more abstraction into implementations to improve reusability and maintainability. The limitation of reusing components is that the user is usually responsible for ensuring that the components work correctly in the new problem.

Reusing solutions when problems are equivalent characterizes much work in theoretical computer science. Once a solution to a problem is found, that solution can be reused in new domains by identifying similarities between problems. For instance, solutions to the problem of graph coloring can be applied to allocating registers in compilers (*cf.* [ASU86]). Where an extensive analysis of a problem is available, this sort of reuse is very important and useful. However, it is not always easy to find similarities between problems, and only problems with very well-defined solutions can be solved in this way.

Reusing solutions on analogous problems characterizes much of the research on machine learning (*cf.* [MCM83, MCM86]). Carbonell [Car83] describes transforming solutions to improve problem-solving. Carbonell identifies a set of operators which can be used to transform previously-obtained solutions to solve portions of new problems. Dershowitz and Manna [DM77] apply similar techniques to programming. Given a program, its specification, and a modified specification, the system computes transformations between the specificiations and applies them to the program to obtain a new version. A difficulty with this is that the system must then prove that the resulting program meets the specification. A related difficulty is that sophisticated reasoning is needed to ensure the new program is executable. For instance, transforming a program to compute square roots into one which finds the position of an element in a sorted array leads to the test '$\mathsf{pos}(A, b) \leq z + y$' where $\mathsf{pos}(A, b)$ denotes the position of b in A. Since pos it the value being computed, the test is not executable; it must be rewritten to '$b \leq A[z + y]$'. In general, the system must have an extensive set of rules to ensure that the transformed programs are executable. It is not clear that such an approach scales up to large problems.

Reusing the *process* of constructing solutions characterizes the sorts of systems this research is concerned with. Given a sequence of steps used to solve a problem, a system can reapply the steps to new problems to obtain new solutions. In some cases, the steps are blindly repeated until they fail. In other cases, the system modifies and reorganizes the sequence of steps to specialize it to the new problem. But in any case, a system examines the inferences used in one problem and applies them to other problems. As a result, the system can rely on the underlying theorem prover to maintain correctness, so it should be easier to scale it up to large problems.

Systems which reuse processes can be categorized by their flexibility. At one extreme are the systems based on *learning macro-operators*; that is, operators which combine the effects of primitive operators [DM86, FHN72, LRN86, MKKC86]. For instance, in explanation-based learning [DM86, MKKC86], the system constructs a proof for a specific example and then replaces the constants in the proofs by variables to obtain a maximally general rule. This allows the system to reuse the proof on similar problems, avoiding search. [HA88, MMS85] apply this method to circuit design, while [FH94, KW94] apply it to theorem proving. However, such rules can be applied only when a specific set of preconditions are satisfied. If any one of the preconditions is violated, such rules provide no guidance on how to proceed.

A more flexible approach is to *reuse proof outlines*; that is, to *replay* them. In this approach, the system stores key steps in developing proofs and replays them to create new proofs. One of the earliest systems to do this was [Wil83]. Reusing proof outlines is more flexible than reusing entire proofs because the general steps of a proof may apply where specific rules do not. This allows more differences between old and new problems. It also allows reusing portions of proofs by replaying subsequences of steps from outlines.

More flexible yet is *derivational analogy* [Car86, Vel92]. In this approach, *all* information related to proof development is stored and made available for reuse. This includes complete sets of rules used to construct proofs, why specific rules were applied, and why alternative approaches were rejected. This maximizes flexibility by providing all information that might be useful to determine if problems are related and deciding how to recover from failures. This method is also based on replaying steps, but replay exercises greater control on how and which steps are applied. However, there is a lot of information to record when constructing a proof and process during replay. Also, using such detailed information means that replay depends strongly on the specifics of the underlying proof development system. This makes implementing derivational analogy difficult.

This thesis discusses replaying proof outlines. We show that replaying proof outlines provides adequate flexibility without requiring the extensive analysis applied in derivational analogy. The result is a mechanism for reusing proofs which is reasonably easy to implement, efficient, and useful.

While the usefulness of replaying proof outlines has been discussed for some time, implementations have appeared only recently. These can be classified by how search is controlled during proof development. In many cases, the underlying proof development system is *fully automated*; that is, it uses domain-specific knowledge and general strategies to control search to minimize the amount of input from the user. This characterizes many theorem provers from the 60's and early 70's [Lin88]. In automated systems, replay is useful for *speedup*: solutions can be obtained without replay, but replay simplifies search by allowing the system to reuse previous work. Such systems include LP [Sil86] which uses replay to help solve for unknowns in symbolic algebra, and APU [Bha91] and DMS [Bax90, Bax92] which incorporate replay into automated programming.

In contrast, many recent systems are *interactive*. These contain little search-control knowledge and rely on the user to develop proofs, such as [Con86, Gor88, Pau86, Red88a]. In these systems, replay is used to *increase automation*. That is, replay simplifies proof development for the user by providing problem-specific search control. Interactive systems incorporating replay include POPART [Wil83], XANA [MF89], and [Gol90]. These systems apply replay to constructing programs. Closely related to these are Redesign [SM85] and BOGART [MB87, Ste87] which apply replay to designing circuits.

The system described in this thesis is based on replaying proof outlines in an interactive environment. Our work differs from previous work in that it is a complete implementation of a general, robust replay system. In contrast, POPART does not address robustness. The user is responsible for constructing robust scripts. Redesign and BOGART address robustness partially, but remain sensitive to relatively small changes in specifications.[3] XANA applies replay to a limited domain: implementing generate-and-test algorithms. This allows replay to be specialized for particular types of problems. [Gol90] applies replay to general-purpose programming, but while it presents methods for improving robustness, the actual implementation did not include those methods. Thus the replay system described in this thesis represents one of the first implementations to address robustness while supporting a wide range of programming problems.

1.4 Thesis Overview

This thesis can be divided into four major parts. The first part, Chapters 2 and 3, discusses the design of a robust replay mechanism. This represents our experience in integrating replay into the Focus transformational implementation system [Red88a, Red90a]. Chapter 2 describes the design issues and how they have been resolved in previous work. Chapter 3 makes these issues concrete by discussing Focus and its associated replay system, ReFocus.

The second part, Chapters 4, 5, and 6, addresses the most difficult, and important, part of replay: how to modify derivation histories so that reexecuting them does not fail. Our solution is based on higher-order generalization. While first-order generalization and higher-order unification have been well-understood for a number of years, higher-order generalization has not been studied extensively. We present a definition for the second-order case and show

[3] See [Mos89] for an analysis for all three of these systems.

that it is computable. This leads to a general definition of syntactic analogy which is useful in domains with rich term structure. Chapter 4 discusses the factors which influence how analogies should be constructed in replay. Chapter 5 defines second-order generalization, gives an algorithm to compute it, and describes how it can be used to build syntactic analogies. This definition of second-order generalization the core contribution of this research. Chapter 6 considers the practical issues of integrating second-order generalization into a replay system. It identifies a useful subset of the generalizations, gives an efficient algorithm for computing them, and illustrates their use in ReFocus.

The third part, Chapter 7, proposes a solution to the problem of deciding if replay is successful. This technique is based on a novel application of term orderings. It explains how term orderings can be used to obtain a discriminating test for deciding if the results of replay meet the user's goals.

The final part, Chapter 8, briefly describes our implementation of replay, ReFocus, and gives examples of applying ReFocus on a number of problems. These include both examples for which replay is successful and examples for which it is not. This helps establish the capabilities and limitations of ReFocus. Chapter 8 concludes with a discussion of the usefulness of replay in proof development systems.

1.5 Contributions

This thesis contributes to the field of software engineering and artificial intelligence in the following ways:

- It demonstrates the feasibility, within limits, of using the transformational implementation paradigm for program maintenance.
- It demonstrates that an extensive, domain-specific knowledge base is not necessary to build a useful replay mechanism.
- It defines second-order generalization and gives an efficient algorithm for computing an important class of generalizations.
- It uses second-order generalization to give a definition of analogy based on the structures of terms.
- It applies second-order analogies to replay.
- It describes an acceptability criterion that is useful for replaying program derivations.

The core contributions are the first and third items. We demonstrate that a robust replay system can be built. This confirms the claims of a number of researchers. We also define second-order generalization. This gives us a tool that has many potential uses beyond replay.

Chapter 2

Replay Design Issues

In the transformational implementation paradigm, the user (*i.e*, the system developer) writes a high-level specification and derives an efficient program. When the specification is changed, the modifications can be propagated to the program by using replay to reexecute each step in the derivation history. Several problems can arise when reexecuting a step:

- It may fail because replay is not able to find the part of the new specification corresponding to a subterm stored with the step.
- It may fail because one of the step's preconditions is not satisfied.
- It may appear to succeed but not produce acceptable results.

A replay system must address all of these problems.

The primary design issue for a replay system is that it must do as much as possible with as little involvement from the user as possible. Replay must be *robust*: in spite of changes to the specification, replay should repeat as much of the derivation as possible. If replay fails too often, the user spends too much time repairing derivations. Replay should also be *autonomous*: it should not require auxiliary information from the user. Otherwise the user may find it simpler, and perhaps even faster, to reconstruct the derivation manually.

This chapter elaborates on these issues and discusses general approaches to making replay robust and autonomous. In particular, it looks at recording derivation histories, propagating differences, checking results, and handling failure. In each case, we discuss general ways to improve support for maintenance as well as how previous systems have approached the problems. Although our focus is on software development, the following discussion also draws from existing replay systems in component design and theorem proving since they must solve similar problems. The specifics of our solutions are postponed to Chapter 3, which presents the Focus transformational implementation system and describes how replay interacts with it.

2.1 Representing Designs

In transformational systems, the design is represented by the derivation history. Each step in the history indicates what decisions were made in implementing the system. For instance, a step might introduce a function which combines the results of two other functions to avoid generating an intermediate result. How well the derivation history captures the design decisions affects both how easily it can be replayed and how useful it is as documentation.

An important issue in derivation histories is the *grain-size*, or generality, of the recorded steps. A step is *small-grained* if it refers to the particular rules to be applied or contains detailed descriptions of where to apply them. Steps which do not store these sorts of details are *large-grained*. For example, "Unfold reduce('MAX, 3) using rewrite rule REDUCTION-OP-TO-FOR-LOOP" [Gol90, p. 116] is small-grained while "achieve recursion" and "simplify" are large-grained.

In replay, there is a conflict between providing guidance and being overly sensitive to changes. Recording small-grained steps usually makes replay more efficient because fewer alternatives need to be considered. However, it also makes replay more *brittle*: small changes in the rules or specifications can invalidate entire sequences of steps. On the other hand, recording large-grained steps means that replay is robust at the expense of recomputing results which may already be available. Thus there is a trade-off between efficiency and autonomy: storing large-grained steps makes replay less efficient but also decreases the chance that the user must repair the derivation by hand. Since our goal is to increase the amount of automation, minimizing the amount of work on the part of the user is the more important issue.

Most implemented replay systems, such as [MB87, MF89, SM85, Wil83], store small-grained steps. In many cases, this is appropriate since rule-based systems use replay for efficiency rather than to increase automation. However, there has been some discussion of storing large-grained steps. In particular, Balzer [Bal85] and Fickas [Fic85] discuss improving replay in the transformational system PADDLE by storing more general steps with an emphasis on recording the goals achieved by each step. Also, Silver [Sil86] discusses storing large-grained operators, such as "isolate," "collect," and "factor," in his equation-solving system LP. The reports on both LP and PADDLE suggest that replay can be made more robust by storing large-grained steps. This is also the approach taken in Focus.

In contrast, HOL [Gor88] stores very small-grained operators. As an example, the proof history in HOL[1] for

$$\forall l_1 : \texttt{list}(*), l_2 : \texttt{list}(*). \, \texttt{LENGTH}(\texttt{APPEND}\; l_1\; l_2) = (\texttt{LENGTH}\; l_1) + (\texttt{LENGTH}\; l_2)\text{''}$$

[1]As given in the HOL88 standard distribution.

```
GEN_TAC
 THEN GEN_TAC
 THEN INDUCT_TAC
 THEN ASM_REWRITE_TAC[MULT_CLAUSES;ADD_CLAUSES;SYM(SPEC_ALL ADD_ASSOC)]
 THEN REWRITE_TAC[SPECL["m:num";"(p*n)+n"]ADD_SYM;SYM(SPEC_ALL ADD_ASSOC)]
 THEN SUBST_TAC[SPEC_ALL ADD_SYM]
 THEN REWRITE_TAC[];;
```

Figure 2.1: HOL proof that * distributes over +.

is

```
LIST_INDUCT_TAC THEN ASM_REWRITE_TAC [LENGTH;APPEND;ADD_CLAUSES];;
```

The first step, `LIST_INDUCT_TAC`, invokes induction to obtain cases for l_1 = `NIL` and l_1 = `CONS` h t. The second step, `ASM_REWRITE_TAC`, rewrites both cases by applying the inductive hypothesis, the definitions of `LENGTH` and `APPEND`, and various theorems of addition. These steps are very fine-grained; all of the names refer to specific rules and tactics in the database. Were the user to rename (say) `LIST_INDUCT_TAC` to `INDUCTION_OVER_LISTS`, replay would need to redetermine which rules to apply in a new proof.

Even more significantly, proofs at this level of detail rarely share a common structure. Consider the corresponding theorem for the distributivity of multiplication over addition:

$$\forall mnp.p * (m + n) = (p * m) + (p * n)$$

Since these two theorems are similar, one would expect their proofs to be similar. However, the HOL proof, given in Figure 2.1, is very different. Even without a detailed explanation of this proof, it is obviously very different from the proof for `LENGTH(APPEND` l_1 l_2). Research has demonstrated that HOL is a very capable theorem prover [Gor88], but incorporating replay into HOL would be difficult.

2.2 Propagating Differences

Propagating differences[2] from specifications to implementations is an integral part of the transformational implementation model. In many cases, the differences are automatically propagated by the rules. For instance, the rule '$A \wedge A$ rewrites to A' applied to '$\mathsf{F(x)} == \mathsf{G(x)} \wedge \mathsf{F(x)} == \mathsf{G(x)}$'

[2]The term 'propagating differences' is used to refer to updating a derivation to apply it to a new problem. This is in contrast to the term 'propagating changes' which refers to using replay to change an implementation after making changes to its specification. Difference propagation arises in both change propagation and derivation-by-analogy.

would automatically propagate any modification made to the subterm F(x) == G(x). In other cases, however, replay must propagate differences explicitly. For example, the definition of an auxiliary function may need to be changed to reflect renamed functions. Replay may also need to change which rule is applied by a step (perhaps because the original rule is no longer available), though doing so is difficult since it requires understanding why specific rules were used. Also, if replay compares the new results against the original results, the original results may need to be updated to correspond to expected differences. This list is certainly not complete; the need to propagate differences is pervasive in replay.

When modifications are very extensive, propagating differences is as difficult as building new derivations. Since propagating differences reflects the need to predict results without the benefit of actually deriving them, minimizing the number and size of references stored in the derivation history improves robustness. This is another reason to store large-grained steps.

A number of systems provide some type of difference propagation. This is partly because most reported systems store small-grained steps in histories; if any references to the specification stored with the steps are not changed, then reexecuting them is likely to fail. Thus these systems depend strongly upon explicit difference propagation. One system to do sophisticated difference propagation is XANA [MF89]. XANA is the replay component of DIOGENES, a system for specializing generate-and-test algorithms. XANA forms correspondences between old and new objects by identifying how they were created. Suppose step S_1 in a derivation creates an object used by step S_2. To find the corresponding object while replaying step S_1, the system determines which object was created when step S_2 was replayed. This is useful when the source of an object is clearly identifiable, but accurate identification is not always possible. When the interactions between the rules become more complex, tracing the source of each object becomes much more difficult.

Goldberg [Gol90] describes providing difference propagation in the transformational implementation system KIDS. KIDS modifies programs by applying a series of tactics. Tactics are rules for doing such operations as combining loops or simplifying expressions. Each tactic is of the form

> *tactic-name* (*parameter-list*) =
> **let** *identifier-list* **in** *tactic* **returns** *identifier-list.*

During replay, the system maintains a list of parameter correspondences which contains the original and new values for each parameter referenced in a tactic. If the value of a parameter is not given by a previous step (perhaps because it was specified by the user during the original derivation), replay checks the parameter correspondence list. If the parameter is not in the list, the system searches upwards in the new abstract syntax tree for a node which contains a parameter which *is* in the list, and then traverses back down the original abstract syntax tree to locate the matching subterm. Goldberg suggests that this method has the potential to be successful because "components are recursively divided into subcomponents, and this parts hierarchy can be used to find corresponding components." However, as in XANA, these heuristics are applicable only to the extent that systems can be divided into discrete components. While some systems permit division, our experiences have shown that some programs can be synthesized only by manipulating interacting components. The parameter correspondence heuristic had not been implemented at the time [Gol90] was written, so there is no discussion of its usefulness in practice.

Our solution is based on building analogies via higher-order generalization. This is described in Chapters 4, 5, and 6.

2.3 Checking Results

In transformational implementation systems, there are three types of failures to contend with:

- the resulting program is not correct,
- some step in the derivation cannot be replayed, or
- the resulting program is not efficient.

Correctness, or how to write code so it matches the specification, is the traditional programming issue. However, this is usually not an issue in replay. Ensuring that the final result is correct is the job of the transformation implementation system. Because this type of system uses equivalence preserving rules, the result of a derivation should always be consistent with the specification automatically. Hence there is no need for replay to check correctness.

On the other hand, replay can and must handle the second problem, failing to apply a step. While this problem is easy to detect, deciding how to proceed is difficult. Recovering from failed steps is discussed in the next section.

Detecting the third problem, failing to create efficient programs, is more difficult. When the user builds a derivation, the goal is to transform the inefficient specifications into a program which meets some criteria such as using a bounded amount of space. Replay fails if the results do not meet the user's goals. In this case, the user must repair the derivation by adding new steps (possibly after establishing more properties about the specification).

It is important that a replay system provides automatic checks for goal failure. If it does not, the user must either check the results by hand or blindly trust replay to produce acceptable results. Checking results manually is tedious and error-prone because it requires examining the logic of the program in detail. Sometimes, the difference between an efficient and inefficient program can be as slight as transposing a pair of function names. Making the user responsible for finding such failures negates much of the usefulness of replay—and possibly of transformational implementation systems—because rechecking can be as time-consuming as using traditional methods to make modifications.

Acceptability is easier to determine in systems which store small-grained steps. In these, being able to apply a rule is usually enough to guarantee that the results will be acceptable, and so the user needs to be warned only when a rule fails to apply. However, detecting failure is more difficult when applying large-grained operations such as "rewrite by all rules." When there is no detailed record of which rules were applied and the effects of each, the system cannot recognize when an important rule has not been applied. Unfortunately, small-grained steps make replay brittle, so storing such steps is not a good solution to the problem of checking results.

One approach to testing acceptability is to simply compare the original and new programs. The assumption is that any change in the final program may mean that it no longer meets the user's needs. Unfortunately, requiring exact matches wastes the user's time on trivial differences. It should be possible to be more selective about which changes are brought to the user's attention. An improvement on this method is to use some form of analogy to directly revise the final program to reflect the changes in the specification. That is, replay could *predict* the form of the derived program and compare the predicted form against the actual results.

There are difficulties with prediction. First, the available methods for finding analogies cannot accurately predict the form of entire programs. It is much easier to predict the form of small terms since there are fewer details and so fewer opportunities for mistakes. Secondly, if the modifications made by the user consist of removing and adding property rules (as opposed to function definitions), it is unclear that there is any better way of predicting the final form than by actually applying the rules. Finally, and most importantly, expecting an exact match against a predicted result fails to recognize the *purpose* of a derivation. That is, the assumption that the user should be informed of *all* differences is false. There is no need to warn the user of inconsequential differences. Ideally, replay would warn the user of potential problems only when it cannot ensure that the goals of the derivation have been met. Generating too many warnings is nearly as bad as not checking at all since in either case the user spends too much time reexamining results.

The problem with detecting goal failure is in determining what the goals are. The system could require the user to state them explicitly, but there is no adequate language for describing most goals. While it is simple to test for some goals, such as completing a proof, it is not easy to test for most goals. For instance, usually the goal of a derivation is to decrease the space or time needed to compute a result. But while criteria for such goals can be stated formally, there is no general algorithm for deciding if a program meets the criteria. For other goals, such as readability,[3] there are no known ways to quantify the criteria. One solution would be to restrict the sorts of criteria that can be used and provide a language for describing only these. But then the user may be interested in some unanticipated goal. For example, [Red88b] describes a derivation which increases the parallelism of an implementation; this is not an obvious goal of a derivation. Therefore, having the user describe the goal of a derivation is not likely to be successful. Heuristics are needed for inferring goals automatically.

Checking goals has not been discussed extensively in the literature. One system that does address goal checking is Silver's LP [Sil86]. Since the purpose of LP is to solve algebraic equations, the goal is to transform equations into the form $x = a$ where x does not appear free in a. As with theorem proving, this provides a simple test for success. However, it is not general enough for program transformation. The other system to address the issue is POPART [Wil83]. It has very limited goal tests: the user can write templates for comparing against the results of key steps during replay. This is not a practical solution if the user intends to build many derivations.

Our solution, as described in Chapter 7, is to record key function calls that are removed during derivation. This captures goals only indirectly, but it is effective on many examples. Combining it with difference propagation makes it more effective because then the recorded function names can be changed. Since there are fewer chances to be wrong when changing small expressions, this is a significant improvement on attempting to predict the form of entire programs.

2.4 Handling Failure

Replay systems must recover from two types of errors: failing to apply a step and failing to derive an acceptable implementation. Recovering from the second type is straightforward: the best strategy is to simply ignore the problem and continue with the next step. If the transformation system is capable of building derivations automatically, then invoking this feature to repair the problem may be helpful. But in systems which are guided by the user, it is better to warn the user than attempt repair so as to avoid wasting time on unprofitable search paths. Furthermore, the user's goals for a derivation may change or be identified incorrectly, in which case there may be no need for repair. More sophisticated sorts of error recovery require heuristics about constructing derivations; such as those described in [Roe92].

On the other hand, failing to apply a step cannot be ignored. Since each step usually establishes the preconditions of the next, one failure is likely to cause others. Some action should be taken to control this behavior.

[3] Of course, this goal is not generally useful in the transformational implementation paradigm.

A very simple system would halt on the first failed step. This is akin to compilers halting at the first syntax error in a program. Programmers prefer compilers which recover from syntax errors so that many can be found in a single run. Likewise, replay systems should continue after a step fails to help locate other problems. Transformational systems are generally slow, so it is useful to design replay so that it can run without any input from the user even when a step fails.

Recovery after a step fails can take several forms:

- postponing a step until it becomes applicable
- skipping a step and continuing
- inserting a new step
- replacing a step with a different one

Ultimately, deciding what action to take is as difficult as building new derivations. If there is no system for deriving programs automatically, the best choice depends upon the type of operation that failed. For example, if a rewrite step unexpectedly generates multiple results, replay can add a step to pick the result which is closest to the expected form. An alternative to recovery is to simply halt; this is the most appropriate action when it becomes apparent that the original derivation is no longer applicable. In such cases, continuing accomplishes little more than to generate a large number of error messages. However, deciding when derivations are not applicable is also difficult.

Most reported systems either halt on errors or skip to the next step. LP [Sil86] is the only system for which sophisticated failure recovery is described. It could add, substitute, or omit steps as necessary. This is because LP is applicable to a limited domain (solving algebraic equations), so Silver and his coworkers were able to develop an extensive theory for solving problems in this domain. Recovering from failed steps in transformational implementation systems is more difficult because the domain is essentially unconstrained.

2.5 Previous Work

In his extensive discussion of replay, Mostow [Mos89] identifies several design issues in replaying derivations:

1. *Representation*: How to represent the original derivation and what information should be included.
2. *Acquisition*: How to capture the derivation. In some systems the user explicitly enters it as a separate step, but in most the user's actions are recorded automatically. Ideally,

systems which record steps would also record the reasons why each step was chosen, but this is an unsolved problem.

3. *Retrieval*: How to choose the stored derivation which is most relevant to the current problem. This is difficult because it requires being able to predict the effects of a derivation on a new problem.

4. *Correspondence*: How to match elements of the original derivation to the current problem.

5. *Appropriateness*: Deciding which steps should be used. The definition of "appropriate" depends upon the capabilities of the system: in a given situation, a step may be *executable* (the preconditions for the step are satisfied), *correct* (the step preserves correctness), *successful* (the step achieves its original purpose), or *desirable* (the step is still better than any of its alternatives).

6. *Adaptation*: Altering the original derivation so it solves the current problem. This means updating, replacing, or removing steps which are not applicable in the new problem.

7. *Partial reuse*: Using portions of the original derivation instead of the whole. This can mean either reusing some continuous subsequence of the steps or picking out only those individual steps which are needed.

It is instructive to examine these design issues in the context of the previous sections. In particular, many of the issues are only relevant in rule-based systems.

1. *Representation*: This is partially addressed by the discussion on derivation histories where it is argued that the representation should consist of large-grained steps. However, we did not discuss explicitly representing the reasons a user made a particular choice. It is not clear how to capture the reasoning without a model of the user; this implies that capturing reasoning is more appropriate in rule-based systems.

2. *Acquisition*: Our approach is very similar to most other systems: the user operations are recorded as they are invoked.

3. *Retrieval*: It is not clear that automated retrieval is useful in interactive systems. When using replay to automate maintenance, the derivation is already chosen. When using replay for derivation-by-analogy, it is not obvious which example should be applied to the problem. Often problems and solutions with very similar structures come from very dissimilar domains.

4. *Correspondence*: This issue was addressed in the section on propagating differences, and seems to be a key problem in replaying derivations in interactive systems.

5. *Appropriateness*: In interactive systems, there is no list of alternative actions, so it is useful to simply replay each step and attempt to recover from failure rather than try to predict whether an operation will be successful. However, determining if a step is successful is important to interactive systems.

6. *Adaptation*: This is also a feature of rule-based systems, though replay may be able to recover from failed steps by adding or changing steps reactively.

7. *Partial reuse*: In an interactive system, partial reuse is limited to replaying subsequences of derivations. More sophisticated partial reuse, such as selecting individual steps to replay in arbitrary order, requires rule-based reasoning.

Thus in interactive systems, the important issues are representation, correspondence, and appropriateness, where appropriateness is understood to be in the sense of determining if a step was successful. The remainder of this thesis addresses these issues for semi-automated proof development systems and for Focus in particular.

Chapter 3

Focus

Focus is a transformational implementation system based on the *fold/unfold* methodology of Burstall and Darlington [BD77]. It provides the basis for the replay system presented in this thesis, ReFocus. Since much of the robustness and autonomy of ReFocus is derived from Focus, this chapter briefly describes the Focus system and gives an example derivation. For a more formal and complete description of Focus, see [Red88a] and [Red90a]. This chapter then discusses Chapter 2 in the context of Focus and presents the basic design of ReFocus. The final section illustrates using ReFocus to reimplement a specification after changes.

3.1 Focus

Focus stores derivations as trees which are manipulated by the tree editor Treemacs [Ham88]. Functions and properties are specified in *program* nodes. Other nodes hold *focus expressions* which are either equations defining new functions or properties the user intends to prove. When the user wishes to unfold a focus (*expanding* it, in Focus terminology) to do case analysis, a subtree is created to hold the various cases, each of which is in turn another focus.

As an example, consider the following function to gather the leaves of a tree. The user first creates a node containing the relevant declarations given in Figure 3.1. Each definition is a conditional rewrite rule of the form $l\,\{c\} \rightarrow r$. This means that when c holds, l equals r and any occurrence of l should be replaced by r. The "." operator is a list constructor similar to the cons function of Lisp as shown by the type declaration of List. Other type constructors are denoted by identifiers starting with capital letters. Functions and variable names start with lower case letters. Focus also supports property rules; these are specified in a *properties:* section of a program node using the same form as definitions. Properties can be proven using Focus, but they are often stated explicitly for expediency.

Focus has four basic operations. The first is to introduce a *focus specification*; this expression is either a property to be proven or a definition of a new function. The second is *simplify*;

```
data List(α) = Nil | α.List(α)
data BinTree(α) = Leaf(α) | Tree(BinTree(α), BinTree(α))

function append :: (List(α), List(α)) → List(α)
function fringe :: BinTree(α) → List(α)

definitions:
    append(Nil, l)             → l
    append(a.x, l)             → a.append(x, l)
    fringe(Leaf(x))            → x.Nil
    fringe(Tree(left, right)) → append(fringe(left), fringe(right))
```

Figure 3.1: The definition of fringe.

it rewrites a term according to the definition rules. Simplification is a form of symbolic evaluation. The third is *rewrite*, which applies all rules, including both definitions and properties. Simplify and rewrite apply the rules until none is applicable; this reduces a term to its *normal form*. Simplification is more efficient than general rewriting, so it is usually better to simplify terms before rewriting them. The fourth operation is *expand*. This is similar to Burstall and Darlington's *unfold* operation. Expand is an inductive operation, while simplify and rewrite are not. Expansion instantiates subexpressions in all possible ways according to the definition rules. Expansion also introduces an inductive hypothesis formed from the expanded term; applying this rule during rewriting implements Burstall and Darlington's *fold* operation. Focus is built so that once a term has been expanded, any application of the inductive hypothesis is valid because it is necessarily applied to a smaller term.

Returning to the fringe function, it is inefficient because of a second traversal of the list during calls to append. These calls can be eliminated by merging the fringe and append functions. The key is to introduce an accumulating parameter for fringe. This is done by introducing a focus specification:

```
flatten(tree, accum) = append(fringe(tree), accum)
```

where the type of flatten is

```
function flatten :: (BinTree(α), List(α)) → List(α)
```

Applying the above operations gives the derivation shown in Figure 3.2. Observe that it is organized as a tree with indentation indicating parent-child relationships. In this figure, each node in the tree has two parts: the initial state (either as the user entered it or as the result of an expansion) and the final state. To derive the program, we need the associative property of append:

Focus: flatten(tree, accum) = append(fringe(tree), accum)
flatten(tree, accum) = append(fringe(tree), accum)
ind. hypothesis: append(fringe(tree), accum) → flatten(tree, accum)

cases from fringe(tree):

1. *case* tree == Leaf(x):
flatten(Leaf(x), accum) = append(x.Nil, accum)
flatten(Leaf(x), accum) = x.accum

2. *case* tree == Tree(left, right):
flatten(Tree(left, right), accum) =
append(append(fringe(left), fringe(right)), accum)
flatten(Tree(left, right), accum) = flatten(left, flatten(right, accum))

Figure 3.2: The derivation of flatten.

append(append(u, v), w) → append(u, append(v, w))

This property is easy to prove using Focus, but we will not do so here to for brevity. The first step in deriving a program for flatten is to expand (unfold) it based on the definition of fringe, giving cases for tree == Leaf(x) and tree == Tree(left, right). As shown, the right-hand side of the first case can be simplified and rewritten to x.accum and the second to flatten(left, flatten(right, accum)). This second step makes use of the associative property of append and the inductive hypothesis. After "closing" the focus node to freeze the derivation and make the new rules available to the rest of the tree, we obtain the program

```
flatten(Leaf(x),            accum) → x.accum
flatten(Tree(left, right), accum) → flatten(left, flatten(right, accum))
```

This is an improvement because the calls to append have been removed.

In addition to the initial and final states, focus nodes store *scripts* which record what operations were applied at each node. Scripts are the primary input used by ReFocus to replay derivations. The entire derivation history for flatten, including scripts, is given in Figure 3.3.

Besides the names of the steps, script entries can also store arguments and other details that are needed during replay. Line (1) in Figure 3.3 is an example of storing arguments. In this instance, the append subterms were explicitly rewritten to force Focus to apply the associativity rule before introducing the call to flatten. Other operations which might be stored in the scripts include case analysis based on the form 'p or not(p)', introducing abstractions (such as changing 'sin(y) + sin(y)' to 'LET x = sin(y) IN x + x'), and picking a particular result when rewrite generates multiple ones. All operations are stored in scripts automatically by Focus. In some cases, Focus also stores information about what was accomplished by the derivation. These are given as multisets stored in the script (such as {flatten $\not\succ$ append, flatten $\not\succ$ fringe}), and will be discussed in more detail in Chapter 7.

The steps for proving properties are very similar. For example, Figure 3.4 gives a derivation of the property

$$\mathsf{fringe(tree)} \rightarrow \mathsf{flatten(tree,\ Nil)}$$

This property allows occurrences of fringe to be replaced by flatten in later derivations. The steps used in the proof are the same as in deriving a program (that is, a subterm is expanded and each case rewritten). The primary difference is that the goal is to reduce each case to an identity. Since both cases are of the form $x = x$ in Figure 3.4, the property is considered to be proven.

3.2 Replay in Focus

This thesis presents the replay system we have added to Focus, *ReFocus*. During replay, ReFocus creates a new derivation (in the form of a tree) mirroring the original (or *prototype*) derivation. It uses a depth-first traversal of the prototype tree, creating corresponding nodes in the new tree when they are needed, copying any specifications from the prototype nodes to the new nodes, and executing the commands stored in the prototype's script. Thus ReFocus is essentially an interpreter for derivation trees. The remainder of this section discusses how the design issues of Chapter 2 are resolved in ReFocus. In particular, we consider grain-size, difference propagation, testing for acceptability, and error recovery.

One of the design principles of Focus is that user-operations should be high-level so that the user is not forced to control the low-level details of constructing derivations [Red88a]. That is, Focus has been designed so that the steps in derivations are large-grained. Operations do not refer to individual rules, and while some operations allow references to specific subterms, these usually take the form of either a single function name or a relatively small term. Building ReFocus so that it uses the same operations makes it easier for the user to understand what happens during replay. It also makes ReFocus more robust.

However, using large-grained operations is not a substitute for explicit difference propagation. For instance, if the user were to change the specification to abbreviate append as ap, the rewrite step (1) in Figure 3.3 would fail. This illustrates that difference propagation is needed for expressions. Likewise, it is needed for the terms stored in the data used for checking results. Finally, difference propagation is needed to update focus specifications. An initial version of ReFocus did not propagate differences, and our experience with using this version confirmed that replay is extremely limited without it.

The mechanics of testing for acceptability are straight-forward. ReFocus compares the final form of each focus expression against the result stored in the prototype derivation. If the expression appears to be unacceptable, ReFocus warns the user of a potential problem. At the leaf nodes of the tree, this checks the final program for acceptability. At the internal nodes, it helps the user determine where the derivation started to fail if the final program is unacceptable. Both tests are done using the methods of Chapter 7.

Finally, we consider error recovery in ReFocus. Two types of errors are generated. The first are those for failing to obtain acceptable results. This signals to the user that the results should be examined carefully. The other type of error is failing to create subtrees. Usually, this is caused by an `expand` step failing to find a variable to instantiate. When this happens, `expand` does not create the cases children paralleling the children in the prototype tree. In this case, ReFocus simply skips the children in the prototype and continues with the rest of the derivation. Usually the user will need to return to the node containing the failed step and repair the derivation by hand.

3.3 A Replay Example

As an example of applying ReFocus, consider modifying the structure of the tree in the flatten example of Figure 3.2. The given function stores information in the leaves of the tree; suppose the user changes it to store information in the internal nodes instead. That is, the user replaces the fringe function by the definition in Figure 3.5. The effect of this change is that the part of the function dealing with the stored information is moved from the base case to the recursive case. For clarity, and to make the example slightly more interesting, we assume the user specifies the following focus equation with the name of the function changed to squash.

$$\mathsf{squash}(\mathsf{tree},\ \mathsf{accum}) = \mathsf{append}(\mathsf{nodes}(\mathsf{tree}),\ \mathsf{accum})$$

where the type of squash is

$$\textit{function}\ \mathsf{squash} :: (\mathsf{InTree}(\alpha),\ \mathsf{List}(\alpha)) \rightarrow \mathsf{List}(\alpha)$$

Secondly, we assume that the user has replaced the property for the associativity of append by the declaration

declare associative append

This allows Focus to use associative matching when rewriting terms involving append.

The flatten derivation can be used as a prototype for deriving a program for squash, as shown in Figure 3.6. Since fringe is replaced by nodes, ReFocus expands the occurrence of nodes in this definition to get the two cases shown. The first case proceeds as before, but the second case is more complicated. Because of the associativity of append, we get two rewrites depending on whether the associative property is applied before or after applying the rule

```
append(nodes(tree), accum) → squash(tree, accum)
```

The resulting alternatives are

i. squash(Node(left, info, right), accum) =
squash(left, info.squash(right, accum))

ii. squash(Node(left, info, right), accum) =
append(squash(left, info.nodes(right)), accum)

The second is less efficient because it retains the call to append. ReFocus chooses the first result since it is closer than the second to what the user accepted in the original derivation. Thus the final program is

```
squash(Tip,                     accum) → accum
squash(Node(left, info, right), accum) → squash(left, info.squash(right, accum))
```

as desired.

3.3.1 Grain Size and Replayability

It is worth noting that replaying this derivation would fail if ReFocus relied on small-grained steps. The sequence of rewrite steps used to transform the right-hand side of the recursive case of flatten is

```
append(append(fringe(left), fringe(right)), accum)
→ append(fringe(left), append(fringe(right), accum)
→ flatten(left, append(fringe(right), accum))
→ flatten(left, flatten(right, accum))
```

where the first step is an application of the associative property of append and the last two are applications of the inductive hypothesis

```
append(fringe(tree), accum) → flatten(tree, accum)
```

In the new derivation, applying the inductive hypothesis (using the associative property of append) the first time gives the term

```
squash(left, append(info.nodes(right), accum))
```

Before applying the inductive hypothesis a second time, the subexpression

```
append(info.nodes(right), accum)
```

must be simplified to

```
info.append(nodes(right), accum)
```

If Focus were to record and reexecute small-grained steps, the simplification step would have to be inserted by ReFocus during replay to obtain the desired result.

3.4 Conclusion

This chapter has described a particular transformational implementation system, Focus, and its replay subsystem, ReFocus. ReFocus inherits much of its robustness and autonomy from the design of Focus. However, it is important that ReFocus compare specifications and apply the differences while constructing the new derivation. For example, if ReFocus we to not change `fringe` to `nodes` before applying the `expand` step in Figure 3.6, the `expand` step would be likely to fail.[1] A harder case arises when picking the right alternative in the recursive case. These examples motivate the need for analogical reasoning, which is the topic of the next three chapters. Chapter 4 discusses what sort of analogical reasoning is needed, Chapter 5 gives a theoretical description for computing useful analogies, and Chapter 6 describes a practical implementation of constructing analogies.

[1]The `expand` operation will pick a default subterm to instantiate, but the default is often inappropriate.

Focus: flatten(tree, accum) = append(fringe(tree), accum)
[*closed with program*:]
flatten(Leaf(x), accum) $\rightarrow$ x.accum
flatten(Tree(left, right), accum) $\rightarrow$ flatten(left, flatten(right, accum))
script:
```
  focus-on-spec()
     {flatten ⊁ append, flatten ⊁ fringe}
  expand(fringe(tree))
```

cases from fringe(tree):

1. *case* tree == Leaf(x):
flatten(Leaf(x), accum) = append(x.Nil, accum)
flatten(Leaf(x), accum) = x.accum
script:
```
  simplify()
  rewrite()
```

2. *case* tree == Tree(left, right):
flatten(Tree(left, right), accum) =
        append(append(fringe(left), fringe(right)), accum)
flatten(Tree(left, right), accum) =
        flatten(left, flatten(right, accum))
script:
```
  simplify()
     {flatten ⊁ append, flatten ⊁ append, flatten ⊁ fringe, flatten ⊁ fringe}
  rewrite(append)                                                          (1)
```

Figure 3.3: The derivation of flatten with scripts.

```
Prove: fringe(tree) → flatten(tree, Nil)
[closed with properties:]
fringe(tree) → flatten(tree, Nil)

  cases from fringe(tree):

  1. case tree == Leaf(x):
     x.Nil = flatten(Leaf(x), Nil)
     x.Nil = x.Nil

  2. case tree == Tree(left, right):
     append(fringe(left), fringe(right)) = flatten(Tree(left, right), Nil)
     flatten(left, flatten(right, Nil)) = flatten(left, flatten(right, Nil))
```

Figure 3.4: The derivation of a property relating fringe and flatten.

```
data InTree(α) = Tip | Node(InTree(α),α,InTree(α))
function nodes :: InTree(α) → List(α)

definitions:
    nodes(Tip)                         → Nil
    nodes(Node(left, info, right)) → append(nodes(left), info.nodes(right))
```

Figure 3.5: Specification of squash.

```
Focus: squash(tree, accum) = append(nodes(tree), accum)
squash(tree, accum) = append(nodes(tree), accum)
ind. hypothesis: append(nodes(tree), accum) → squash(tree, accum)

  cases from nodes(tree):

  1. case tree == Leaf(x):
     squash(Tip, accum) = append(Nil, accum)
     squash(Tip, accum) = accum

  2. case tree == Node(left, info, right):
     squash(Tree(left, info, right), accum) =
             append(append(nodes(left), info.nodes(right)), accum)
     squash(Tree(left, info, right), accum) =
             squash(left, info.squash(right, accum))
```

Figure 3.6: The derivation of squash.

Chapter 4

Analogy and Replay

Analogy is a general mechanism used to transfer knowledge from one domain to another. When used for problem solving, an analogy is formed by comparing the original and new problems. The results of this comparison are used to modify the original solution so that it solves the new problem. In replay, the "problem" is a specification and the "solution" is a derivation used to create an implementation. Pictorially, we have

$$\begin{tikzcd}
\text{original specification} \arrow[r, Rightarrow, "\text{analogy}"] \arrow[d, "\text{original derivation}", name=L] & \text{new specification} \arrow[d, "\text{new derivation}", name=R] \\
\text{original implementation} & \text{new implementation}
\arrow[from=L, to=R, Rightarrow, "\text{applied analogy}"]
\end{tikzcd}$$

Thus, analogy is the mechanism by which differences in specifications are propagated from one derivation to another. This chapter considers the issues of building analogies for use in replay.[1]

Section 4.1 considers the types of differences between specifications that replay may encounter. An important factor in this is how replay is used. When replay is used to propagate modifications, the differences are likely to be relatively minor, such as replacing one function name by another. But when replay is used for derivation-by-analogy, the differences are likely to be major, possibly based on an entirely different set of function names.

The second factor to consider when constructing analogies is deciding how to constrain them. Constraints are needed because analogy can relate nearly any two concepts. Section 4.2

[1]This is in contrast with using replay to implement analogy, as discussed in Chapter 1.

discusses syntactic and semantic constraints and suggests that syntactic ones are more useful in ReFocus.

Section 4.3, gives examples of using analogy in ReFocus. These examples show most of the cases in which analogy is needed. Section 4.4 expands on this discussion.

The examples and discussions in this chapter lead to using second-order generalization to construct analogies. A definition of second-order generalization and algorithms to compute it are given in Chapters 5 and 6.

4.1 Differences in Specifications

The mode in which replay is used—propagating changes or derivation-by-analogy—affects the sorts of differences between specifications that can be expected to arise. In change propagation, the differences are typically minor; in derivation-by-analogy, they are often extensive. The mode also affects how the analogies are used. In change propagation, typically only a few terms need to be updated to match the new specification. But in derivation-by-analogy, where there is much less in common between the original and new specification, the entire derivation must be updated. However, in spite of the differences, these two modes are closely related: if modifications are extensive enough, there is less in common between the old and new derivations and so the old derivation is used for general guidance rather than as an exact plan.

This section examines the two modes in more detail. It discusses how the modes are used and what sort of differences they generate.

4.1.1 Propagating Modifications

Specifications are modified for a variety of reasons. The most obvious is to correct errors or provide support for new requirements. For example, the user might add another branch to an if statement to account for a new case. This corresponds to the sort of changes made while debugging in traditional programming environments, except that the work is directed towards changing the specification rather than the implementation. However, specifications are also changed during normal development. Using stepwise refinement, some functions are left undefined (or stubbed-out) while the user builds other parts of the system. Once the missing functions are defined, replay is used to integrate them into the rest of the system to provide further improvements in efficiency.

When propagating modifications in specifications, we can expect the changes to be relatively minor. In particular, most symbol names stay constant. However, we should not assume that only one change takes place at a time. It is useful to delay running replay until after a number of modifications have been made throughout a system.

Modifications can be classified as changes in the following:

- *type signatures:* For example, changing

  $$\textit{function}\ \mathsf{lookup} :: \mathsf{int} \times \mathsf{list(int)} \rightarrow \mathsf{bool}$$

  to

  $$\textit{function}\ \mathsf{lookup} :: \mathsf{list(char)} \times \mathsf{list(list(char))} \rightarrow \mathsf{bool}$$

  to change from looking up a key in a list to searching for a name in a table.

- *the order and names of arguments:* For example, changing

  $$\mathsf{exp(x, base)} \rightarrow \ldots$$

  to

  $$\mathsf{exp(base, pow)} \rightarrow \ldots$$

  Often, this kind of change also involves changing types, though in this example both arguments have the same type.

- *the number of cases in definitions:* For example, adding a case to a definition of $\mathsf{nth\_member(n, l)}$ so that it returns Nil if $\mathsf{n} > \mathsf{length(l)}$ rather than signaling an error. This kind of change often reflects a change in types.

- *function and constructor names:* For example, replacing occurrences of min by max. Changing constructor names usually reflects a change in types.

- *the addition of pre- or post-conditions:* For example, adding the stability condition to a quicksort specification.

- *the addition or deletion of subterms:* For example, changing

  ```
  f(a / b)
  ```

  to

  ```
  f(if b == 0 then Infinity else a / b)
  ```

  Often, as in this example, the new subterm forms a context for some subterm which does not change.

The most difficult modification to handle is adding subterms. Consider the above example: If / is used in other contexts, it may be difficult to determine which occurrences should be modified. Some amount of contextual information is needed to disambiguate such transformations.

4.1.2 Derivation-by-Analogy

The derivation-by-analogy mode of replay applies the steps of a solved problem to an unsolved problem. Using replay in this mode saves effort for the user by reusing old designs. For example, the derivation of a tail-recursive implementation of factorial can be used to guide the derivation of a program to reverse a list. Though these functions manipulate different sorts of objects (natural numbers in one and lists in the other), the derivations are similar.

While the differences between specifications are usually minor when propagating changes, they are usually major when using replay for derivation-by-analogy. In particular, identifier names in the two problems are usually different. Thus the system must rely more on matching by position and other syntactic clues.

Another difficult aspect of derivation-by-analogy is that auxiliary function specifications (focus specifications) must be updated so they apply to the new problem. Again, this is more of an issue in derivation-by-analogy than in change propagation because of the differences in the names of functions. If replay does not update auxiliary function specifications, using replay for derivation-by-analogy is certain to fail.

4.1.3 Determining the Replay Mode

It is not always clear when replay is being used for derivation-by-analogy and when it is being used to propagate changes. Types can provide useful hints because they appear to change less when the user modifies specifications. A more pragmatic difference is that when replay is used to transfer knowledge, the old results should be retained instead of replaced. Since automatically replacing results could destroy work, ReFocus provides two commands: one to replay after making modifications and another for derivation-by-analogy. Thus ReFocus essentially leaves it to the user to decide how replay is being used. The same definition of syntactic analogy is applied in both cases.

4.2 Syntactic vs. Semantic Analogies

Analogy is a very flexible mechanism for knowledge transfer because it can relate nearly any two concepts. This makes it very useful in domains for which there is no complete theory. However, the flexibility of analogy also means a system can spend too much time evaluating useless candidates. To make analogy useful, there must be constraints on what can be matched. This section discusses the two most common types of constraints: syntactic and semantic.

Syntactic constraints are based on the *structure* of the objects being compared. The most common constraint is to require matches to be made among objects in the same relative positions. This assumes is that objects in similar positions serve similar functions. For instance, suppose an analogy is being formed between A(B) and C(D). The natural match (from the view of syntax) is

$$\begin{array}{ccc} \mathsf{A} & \leftrightarrow & \mathsf{C} \\ \mathsf{B} & \leftrightarrow & \mathsf{D} \end{array}$$

as opposed to

$$\begin{array}{ccc} \mathsf{A} & \leftrightarrow & \mathsf{D} \\ \mathsf{B} & \leftrightarrow & \mathsf{C} \end{array}$$

The second correspondence is unnatural because it does not respect the relationships between A and B and between C and D.

Semantic constraints are based on the *meanings* of the terms being matched. This assumes that useful matches are ones in which paired objects have similar definitions. For example, it is natural to match max and min because each finds an extremum of a set. Another way to use semantics is by comparing how objects are used in proofs, either in support of or as supported by other inferences.

The division between semantic and syntactic information is not clear-cut. For example, the traditional approach to types suggests that they are semantic, while the categorical view suggests that they are syntactic. Also, semantic information can be reflected in syntax, such as in the definitions of max and min. Often the only difference between the definitions of max and min is that $>$ has been replaced by $<$. For the purposes of this thesis, we use a simplistic distinction: we consider information to be semantic if it is entered by the user primarily for analogical inferences and syntactic otherwise. That is, information is considered to be semantic if it can not be extracted from the theory by the given inference engine.

The following paragraphs examine how syntactic and semantic constrains have been used in previous research on analogy.

Many systems use syntax to generate candidate analogies. Evans [Eva68] uses syntax to generate candidate matches in solving simple geometric-analogy problems. The base case is a pair of figures containing some arrangement of objects. Each figure is represented as a set of terms such as

```
((inside p1 p2) (above p2 p3) (above p1 p3))
```

and

```
((above p1 p2) (above p2 p3))
```

A correspondence is found between the figures, and this is used to build a map from one figure to the other using such relations as *remove* and *translate*. Given a third figure and a small set of potential matches, the system attempts to find the match which can be obtained by applying the original map to the third figure.

More recently, Falkenhainer, Gentner, and Forbus have developed a model of analogy which emphasizes syntactic similarity measures [FFG86, FFG89, Gen83, Gen89]. Their work is designed to understand analogies between physical systems such as the solar system and an atom. These systems are represented using a typed predicate calculus. In this calculus, predicates are divided into two categories: attributes (monadic predicates) and relations (non-monadic predicates). To construct an analogy, attributes are ignored and the remaining predicates matched to generate a set of possible analogies. Attributes are ignored because research suggests that people find them unimportant in analogies [Gen83]. After the initial matching stage, the candidate analogies are ranked on the basis of the "order" of the terms, where the order of an entity is defined to be zero and the order of a predicate is one greater than the maximum order of its arguments.[2] The theory prefers maps which contain consistent matches for higher-order relations. Thus the structure-mapping theory emphasizes syntax.

Other systems rely more on semantic constraints. For instance, Kling [Kli71] uses analogy to help a resolution theorem prover find proofs faster by reducing the search space. In his system, all inference rules are annotated with descriptive properties. Given an example proof, the system finds all clauses in a database which match some step used in the example proof. These clauses are then passed to the resolution theorem prover along with the goal clauses. The annotations impose semantic constraints on the analogy.

McDermott [McD79] presents another way to constrain semantic analogies. His system solves planning problems which arise in a paint shop. Given a new task, the system searches a database for a plan which completes a similar task. Two terms are similar if they have the same ancestor in a type hierarchy. The terms in the goals of each task are compared, and the closest match is chosen. For example, "washing" is like "painting" because both are instances of "spraying," so a plan for painting a table is used to guide constructing a plan for washing a safe. Once an analogy has been elaborated, the system uses a theorem prover to verify that the resulting plan is correct. Since the type hierarchy is a form of definition, this system primarily relies on semantic information.

Both syntactic and semantic constraints have advantages. Syntactic constraints are attractive because they require no added information from the user. However, syntactic constraints

[2]This definition of order is not standard. The more common measure of an order of a term is based on its type. But in this case, the order is actually the *depth* of the term. This has unexpected results. The order of $Greater(x, y)$ is one, while the order of $Greater(sqr(x), y)$ is two. This makes mapping $Greater(sqr(x), y)$ as important as mapping a causal relation such as $Causes(Greater(x, y),\ Warmer(x, y))$.

do not ensure that the matches are sensible. For instance, replacing addition by equality is rarely appropriate.[3] Semantic constraints can reduce the number of such mistakes. However, semantic constraints are more difficult to compute because they rely on a deeper understanding of how objects are used. Alternatively, descriptive properties can be provided so that there is no need for deep analysis. But this limits the analogies that can be found to just those which can be inferred from the descriptions. Most systems use a combination of semantic and syntactic constraints to overcome the problems of each.

Syntactic constraints are particularly appropriate for replay in interactive systems (as opposed to rule-based systems). The extensive structure of terms provides a number of syntactic clues on how to form correspondences. On the other hand, semantic constraints are less appropriate because most interesting objects are defined by the user and so unless the user enters some type of database describing properties of the objects, there is no basis for constraining analogies. Asking the user to explicitly enter such information is not practical since maintaining it just for replay is not worth the effort.

4.3 Replay Examples

Before discussing the details of building analogies, let us examine where syntactic analogy is needed in ReFocus. These examples illustrate why constructing analogies is essential for replay. How to automate this is the subject of Chapters 5 and 6.

The first example continues the example from Section 3.3.

Example 4.1 Compare the derivation of flatten in Figure 3.3 (page 28) against the derivation of squash in Figure 4.1. There are three places where analogy is needed:

- At line (1) in Figure 4.1, the expand(fringe(tree)) operation must be mapped to expand(nodes(tree)).
- The cases must be matched together: tree == Leaf(x) must be matched to tree == Tip and tree == Tree(l, r) to tree == Node(l, i, r). In this example, matching children according to tree position is sufficient. However, this is an accident of how the fringe and nodes functions were defined; if the definitions of one of them listed the recursive case first, then the cases would be in reverse order. Analogy is needed to find the closest matches between the cases so that ReFocus can apply the correct operations to each.
- After the rewrite(append) step in the second case, ReFocus must choose the best match between the original result and the two candidates in the new derivation. This is done

[3]Even in C programs, where such a replacement would be legal!

```
Focus: squash(tree, accum) = append(nodes(tree), accum)
[closed with program:]
squash(Tip, accum) → accum
squash(Node(l, i, r), accum) → squash(l, i.squash(r, accum))
script:
  focus-on-spec()
     {squash ⊁ append, squash ⊁ nodes}
  expand(nodes(tree))                                              (1)

 cases from nodes(tree):

 1. case tree == Tip:
    squash(Tip, accum) = append(Nil, accum)
    squash(Tip, accum) = accum
    script:
     simplify()
     rewrite()

 2. case tree == Node(l, i, r):
    squash(Node(l, i, r), accum) =
             append(append(nodes(l), i.nodes(r)), accum)
    squash(Node(l, i, r), accum) =
             squash(l, i.squash(r, accum))
    script:
     simplify()
         {squash ⊁ append, squash ⊁ append, squash ⊁ nodes, squash ⊁ nodes}
     rewrite(append)
     pick(squash(Node(l, i, r), accum) = squash(l, i.squash(r, accum)))   (2)
```

Figure 4.1: The derivation of squash.

by passing the term to be chosen as a argument to the `pick` command, as shown on line (2). Analogy is needed to find the best match.

The next example shows how analogy is needed to transform lemmas.

Example 4.2 Consider proving a property which states that fringe never returns an empty list:

$$\mathsf{len(fringe(tree)) == 0 \rightarrow False}$$

where len is defined as

$$\begin{aligned}&\mathsf{len(Nil)} \rightarrow \mathsf{0}\\&\mathsf{len(x.xs)} \rightarrow \mathsf{S(len(xs))}\end{aligned}$$

(Note that successor notation is being used to represent numbers: $\mathsf{S(0)}$ represents 1, and so on.) This property cannot be proven without a lemma. In this case, the necessary lemma is

$$\mathsf{len(fringe(t)) + u == 0 \rightarrow False}$$

Proving these two properties by mutual induction is simple in that it requires little further input from the user. The derivation is shown in Figure 4.2a, where the lemma is shown as a subtree of the original property.

This derivation sequence can also be used to show a related property:

$$\mathsf{size(tree) == 0 \rightarrow False}$$

where size is defined as

$$\begin{aligned}&\mathsf{size(Leaf(x))} \rightarrow \mathsf{1}\\&\mathsf{size(Tree(l, r))} \rightarrow \mathsf{size(l) + size(r)}\end{aligned}$$

The derivation proceeds as before (see Figure 4.2b) except that the lemma must be changed to

$$\mathsf{size(t) + u == 0 \rightarrow False}$$

Analogy is needed in order to create the new lemma.

4.4 Analogy in ReFocus

These examples illustrate the primary places where analogy is needed in ReFocus. These situations can be divided into two types of problems: how to transform terms stored in the derivation history, and how to find the closest matches between sets of terms.

Prove: len(fringe(tree)) == 0 → False
[*closed with properties*:]
len(fringe(t)) == 0 → False

> *Prove:* len(fringe(t)) + u == 0 → False
> [*closed with properties*:]
> len(fringe(t)) + u == 0 → False
>
> *cases from* len(fringe(t)) + u:
>
> *1.* *case* u == 0:
> len(fringe(t)) == 0 = False
> False = False
>
> *2.* *case* u == S(b):
> S(len(fringe(t)) + b) == 0 = False
> False = False

cases from fringe(tree):

1. *case* tree == Leaf(a):
len(a.Nil) == 0 = False
False = False

2. *case* tree == Node(l, r):
len(append(fringe(l), fringe(r)))
== 0 = False
False = False

a. Proof of len(fringe(tree)) == 0 → False

Prove: size(tree) == 0 → False
[*closed with properties*:]
size(tree) == 0 → False

> *Prove:* size(t) + u == 0 → False
> [*closed with properties*:]
> size(t) + u == 0 → False
>
> *cases from* size(t) + u:
>
> *1.* *case* u == 0:
> size(t) == 0 = False
> False = False
>
> *2.* *case* u == S(b):
> S(size(t) + b) == 0 = False
> False = False

cases from size(tree):

1. *case* tree == Leaf(x):
1 == 0 = False
False = False

2. *case* tree == Node(l, r):
size(l) + size(r) == 0 = False
False = False

b. Proof of size(tree) == 0 → False

Figure 4.2: Tree size properties.

A number of terms in the derivation history may need to be updated. In many cases, the term is a reference to some part of a larger term that is being operated on, such as the `expand` operation in Example 4.1. These references are usually small, but updating them correctly is critical to the success of replay. In other cases, the term to be updated is the specification of a function or property, such as the lemma in Example 4.2. Updating these is useful because large derivations often involve a number of focus specifications and making the changes by hand is tedious. However, these terms are large, so the analogy mechanism is more likely to fail. Furthermore, they are often subtle because they capture the design intended by the user by strongly constraining the derivation [Red90a]. In a production version of Refocus, radical changes would need to be accepted by the user before continuing with a derivation.

Matching sets of terms is equally important. Several Focus operations, such as `expand`, create cases by instantiating subexpressions. These cases must be matched during replay so that the correct prototype script is applied to the correct case. Likewise, when an operation (such as `rewrite`) generates multiple results, ReFocus must choose the result which is the closest to the result in the prototype.

Another requirement that was not illustrated in the above examples is that ReFocus must determine when the new derivation is completed. This includes checking the final results to ensure that all the goals were achieved. It also includes checking intermediate results so that ReFocus can discover when it is failing. One way to check results would be to use analogy to predict the final form of the program. However, this would be difficult to do accurately. Instead, ReFocus uses information extracted from a user-specified precedence relation. Because this information is based on small subterms, analogy can be used to update them reliably during replay. This is discussed further in Chapter 7.

4.5 Conclusion

In this chapter, we have examined how analogy is useful in replay, what sorts of differences it must accommodate, and what information can be used. In the next chapter, we present a definition of analogy which meets the needs of replay in interactive systems. This definition both generates rules used to transform expressions and gives us a metric for comparing terms to decide which pairs are maximally similar.

Chapter 5

Analogy by Second-Order Generalization

As illustrated in Chapter 4, analogical reasoning is a necessary part of replaying program derivations. There are two primary ways in which analogy is used by replay: to update terms stored in derivation histories, and to match terms between a derivation and its prototype. Nearly every replay operation uses analogy in one or both of these ways.

While the usefulness of analogical reasoning is clear, how to define it is not. Chapter 4 argues that an analogical mechanism for replay cannot rely on semantic information because it is not readily available. Instead, replay must rely on syntactic information, using commonalities between the structures of terms to control what sets of symbols are matched to each other. In this chapter, we use generalization to identify common structure. More specifically, we present a definition of syntactic analogy based on *second-order generalization*. It uses a very flexible representation for terms to capture similarities even when they are embedded in dissimilar contexts. This flexibility is especially important to replay.

Section 5.1 discusses generalization and its role in analogy. Section 5.2 defines generalization using concepts from unification theory. Sections 5.3 through 5.5 consider various representations for terms to maximize the usefulness of the generalization. In those cases where generalization is well-defined, we present algorithms to compute it. Finally, Section 5.6 illustrates using second-order generalization in ReFocus.

5.1 Generalization and Analogy

Analogy can be viewed as a problem in constructing maps between pairs of terms. This section describes constructing analogical maps and explains their relationship to generalization. It also shows that standard definitions of generalization are inadequate. This leads to a discussion of how terms should be represented to support more useful definitions of generalization. This section concludes with a discussion of related work in generalization and a description of the notation used to represent substitutions.

There are two stages to building an analogical map: identifying similarities, and constructing rules from unmatched parts. The first stage is *generalization*: given a pair of instances, find a term which captures as much common information as possible. Variables are inserted as needed to generalize those subterms which are not common to both instances. The output from this stage is a pair of substitutions. For example, consider finding an analogical map from

$$\mathsf{F}(\mathsf{a}, \mathsf{G}(\mathsf{a}, \mathsf{b}), \mathsf{a})$$

to

$$\mathsf{F}(\mathsf{c}, \mathsf{H}(\mathsf{b}, \mathsf{c}), \mathsf{c})$$

One possible generalization is

$$\mathsf{F}(x, f(x, \mathsf{b}), x) \tag{5.1}$$

where sans serif is used to denote constants and *italics* to denote variables. The substitution $\{f \mapsto \mathsf{G}, x \mapsto \mathsf{a}\}$ maps $\mathsf{F}(x, f(x, \mathsf{b}), x)$ to $\mathsf{F}(\mathsf{a}, \mathsf{G}(\mathsf{a}, \mathsf{b}), \mathsf{a})$, and the substitution $\{f \mapsto \mathsf{H}\gamma, x \mapsto \mathsf{c}\}$ maps $\mathsf{F}(x, f(x, \mathsf{b}), x)$ to $\mathsf{F}(\mathsf{c}, \mathsf{H}(\mathsf{b}, \mathsf{c}), \mathsf{c})$, where γ is the commute function which changes u, v to v, u.

The second stage is *map building*: using the substitutions to construct rules which transform one system into another. This is done by matching the bindings of the variables in the substitutions. Pairing the bindings for the variables gives the analogical map

$$\{\mathsf{G} \Rightarrow \mathsf{H}\gamma, \mathsf{a} \Rightarrow \mathsf{c}\} \tag{5.2}$$

Thus the basis of building analogical maps is finding a generalization of the terms being compared.

Since there should not be any transforms for subterms that are shared between two problems, an analogical map should contain as few symbols as possible. This means that a generalization should include as many symbols as possible from each instance term. That is, generalizations should be *maximally specific.* For example, another generalization of $\mathsf{F}(\mathsf{a}, \mathsf{G}(\mathsf{a}, \mathsf{b}), \mathsf{a})$ and $\mathsf{F}(\mathsf{c}, \mathsf{H}(\mathsf{b}, \mathsf{c}), \mathsf{c})$ is

$$\mathsf{F}(x, y, z)$$

This leads to the analogical map

$$\{\mathsf{a} \Rightarrow \mathsf{c}, \mathsf{G}(\mathsf{a},\mathsf{b}) \Rightarrow \mathsf{H}(\mathsf{b},\mathsf{c}), \mathsf{a} \Rightarrow \mathsf{c}\}$$

This map is less useful because it contains a redundant rule and because it suggests G can be replaced by H only when the arguments are a and b. Constructing analogical maps should start with the maximally specific generalization.

The most common definition of maximally specific generalization was first presented in [Plo70, Plo71, Rey70]. Reynolds and Plotkin independently observed that the set of first-order terms along with the relation

$$v \leq u \iff \exists\theta \text{ such that } \theta(u) = v \tag{5.3}$$

and an added top element forms a complete lattice.[1] This ordering is known as the *substitution ordering.* In this lattice, the least upper bound ($\sqcup$) of two terms is given by Robinson's unification algorithm [Rob65]. Plotkin and Reynolds devised a complementary *anti*-unification algorithm to compute the greatest lower bound ($\sqcap$). An elegant version of this algorithm was given by Huet [Hue76, LMM88]:

Algorithm 5.1

$$\begin{array}{rcll} F(\ldots,s_i,\ldots) \sqcap F(\ldots,t_i,\ldots) & = & F(\ldots,s_i \sqcap t_i,\ldots) & \forall F \in \mathcal{C} \\ s \sqcap t & = & \phi(s,t) & \text{otherwise} \end{array}$$

where $\mathcal{C}$ is the set of constants (including function symbols) and ϕ is a bijection between pairs of terms and a set of variables.

Example 5.2

$$\mathsf{F}(\mathsf{a},\mathsf{G}(\mathsf{a},\mathsf{b}),\mathsf{a}) \sqcap \mathsf{F}(\mathsf{c},\mathsf{H}(\mathsf{b},\mathsf{c}),\mathsf{c}) = \mathsf{F}(x,y,x)$$

Where ϕ is

$$\begin{array}{rcl} \phi(\mathsf{a},\mathsf{c}) & = & x \\ \phi(\mathsf{G}(\mathsf{a},\mathsf{b}),\mathsf{H}(\mathsf{b},\mathsf{c})) & = & y \end{array}$$

The function $\sqcap$ defines the maximally specific *first-order generalization* of any two terms. Many systems based on analogy use some variant on this definition [Owe90]. This is appropriate: most domains are represented naturally by terms with simple structure. But in domains where terms have a more complex structure, first-order generalization is less useful. For example,

[1]Assuming that we identify terms which differ only in the names of their variables.

the analogical map for Example 5.2, $\{\mathsf{a} \Rightarrow \mathsf{c}, \mathsf{G}(\mathsf{a},\mathsf{b}) \Rightarrow \mathsf{H}(\mathsf{b},\mathsf{c})\}$, transforms G to H only when the arguments are a and b. In contrast, the map 5.2 transforming G to H is independent of any arguments that are present. This map is obtained from *second-order generalization.* First-order generalization is inadequate because whenever the topmost symbols of two terms differ, the entire terms are replaced by a variable and any similarities among the subterms are lost.

The need for capturing common subterms is particularly acute for program derivations. Consider the problem of finding an appropriate result to pick in the derivation of a program for squash in Section 3.3. We suggested that this would be done by choosing the closest match to

t: flatten(Tree(left, right), accum) = flatten(left, flatten(right, accum))

between

a: squash(Node(left, info, right), accum) = squash(left, info.squash(right, accum))

and

b: squash(Node(left, info, right), accum) = append(squash(left, info.nodes(right)), accum)

One possible method to do this would be to choose the match with the largest first-order generalization. However, both $t \sqcap a$ and $t \sqcap b$ are trivial: $x = y$, so first-order generalization provides no help.[2]

The solution is to choose a representation for generalizations in which variables can be used to represent a *context.* For instance, the f in $\mathsf{F}(x, f(x,\mathsf{b}), x)$ is replaced by G in one case and $\mathsf{H}\gamma$ in the other. This allows common subterms, such as the b, to appear in the generalization even if they are embedded within distinct contexts, This prevents the subterms from appearing in the analogical maps. As shown in the squash example, such spurious information can be a significant problem. In the next section, we consider how to represent terms in which variables abstract context.

5.1.1 Second-Order Generalization with Combinators

To introduce variables which abstract contexts, we must introduce some form of second-order logic. This allows substituting functions for variables. In the simple cases, the function is the name of a constant, such as G. In more complex cases, the function reorders or otherwise manipulates the subterms, such as $\mathsf{H}\gamma$ where γ switches the order of the subterms c and b. To

[2] An alternative representation of terms, such as

$$\mathtt{apply}_2(\mathtt{flatten}, \mathtt{left}, \mathtt{apply}_2(\mathtt{flatten}, \mathtt{right}, \mathtt{accum}))$$

improves the first-order generalization slightly, but the evidence for choosing one match over another is still unconvincing. Changing representations is discussed in Chapter 6.

manipulate subterms, we introduce an equational theory. This section (informally) introduces the term representation and equational theory upon which we base generalization, illustrates using such terms in generalization, and discusses some of the issues of defining second-order generalization.

The standard notation for systems incorporating second-order logic (or, more generally, higher-order logic) is typed λ-calculus [Chu40, Wol93]. However, λ-terms are relatively complex because application is defined by variable substitution: $(\lambda x.r)s = t$ if t is r with all free occurrences of x replaced by s. Equivalently, we can write $f(s) = t$ where f is defined by $f(x) = r$. The difficulty is that there is no control on how s is used within $\lambda x.r$. The variable x may occur once in r, many times, or not at all. In Section 5.4, we show that the lack of restrictions on how variables are used results in second-order generalization not being well-defined.

This motivates turning to notations based on *combinators*. In contrast to λ-calculus, application of combinator terms is defined by symbols (*i.e*, combinators) which control how data is passed from one function to the next. We will show that by restricting the combinators, we obtain a class of terms for which generalization is well-defined.

The combinators we use are those which are motivated by category theory (*cf.* [AL91, Cur93, Mac71]). In particular, we consider two types of categorical combinators: monadic combinators for systems in which all functions have a single argument and cartesian combinators for systems in which functions may have multiple arguments.

We first consider combinators for functions with one argument, *i.e*, monadic functions. Because there is only one way to pass data between monadic functions, the combinators for monadic terms are just composition and identity. Composition is denoted by juxtaposition: if r and s are terms, then their composition is rs. Identity is denoted by $\mathbf{1}$: $\mathbf{1}s = s\mathbf{1} = s$. We call terms made of these combinators *monadic combinator terms*. Thus a term such as $\lambda x.\mathsf{F}(\mathsf{G}(x))$ is written as the monadic combinator term $\mathsf{F}\,\mathsf{G}$, and the equality $(\lambda x.x)\circ(\lambda y.\mathsf{F}(y)) = \lambda y.\mathsf{F}(y)$ as $\mathbf{1}\,\mathsf{F} = \mathsf{F}$. Furthermore, composition is taken to be associative, so we often omit parentheses and write terms such as $\mathsf{F}(\mathsf{G}\,\mathsf{H})$ and $(\mathsf{F}\,\mathsf{G})\,\mathsf{H}$ as $\mathsf{F}\,\mathsf{G}\,\mathsf{H}$. Observe that if we interpret $\mathbf{1}$ as the empty string (usually denoted by ϵ) and composition as concatenation, then monadic combinator terms are essentially strings. The difference is that composition in terms is restricted by types. If the type of t is $X \to Y$, denoted $t : X \to Y$, and $s : Y' \to Z$, then st is defined only if $Y = Y'$.

To support functions with multiple arguments, we could assume polyadic function constants and use a notion of composition for polyadic functions [Lam87]. However, this makes the notation overly complex. Instead, we use some form of a product structure to combine multiple arguments into a single argument. Polyadic functions are then treated as monadic functions from a product type.

The most obvious notion of products is that of cartesian products. We introduce the type symbol $\times$ such that if A and B are types, $A \times B$ is the type representing their product. The combinators involved with product types are the pairing combinator $\langle , \rangle$ and the projection operators π and π'. The term $\langle r, s\rangle$ produces a pair of values and the projection operators decompose such pairs by selecting one of the components. This motivates the equivalences $\pi\langle r, s\rangle = r$, $\pi'\langle r, s\rangle = s$, and $\langle \pi t, \pi' t\rangle = t$. We call terms made of these combinators *cartesian combinator terms.* Cartesian products are used when applying a function to a pair of terms, so $\mathsf{F}(\mathsf{a}, \mathsf{b})$ is written as $\mathsf{F}\langle \mathsf{a}, \mathsf{b}\rangle$ and the equality $(\lambda xy.\mathsf{F}(y))(\mathsf{a}, \mathsf{b}) = \mathsf{F}(\mathsf{b})$ as $\mathsf{F}\pi\langle \mathsf{a}, \mathsf{b}\rangle = \mathsf{F}\,\mathsf{b}$.

In the combinatorial framework, first-order terms can be thought of as "nullary" functions. We also call these *closed terms* to emphasize that they have no (useful) inputs. To distinguish between first-order and second-order terms, we introduce a special *unit type* u and a combinator $!_X : X \to \mathsf{u}$. Any term of type $\mathsf{u} \to B$ is considered a "first-order" combinator term, while terms of type $A \to B$ (for $A \neq \mathsf{u}$) are considered "second-order." Thus Example 5.2 is written in the combinator notation as

$$\mathsf{F}\langle \mathsf{a}, \langle \mathsf{G}\langle \mathsf{a}, \mathsf{b}\rangle, \mathsf{a}\rangle\rangle \sqcap \mathsf{F}\langle \mathsf{c}, \langle \mathsf{G}\langle \mathsf{b}, \mathsf{c}\rangle, \mathsf{c}\rangle\rangle \quad = \quad \mathsf{F}\langle x, \langle y, x\rangle\rangle$$

where, if $\mathsf{a}, \mathsf{b}, \mathsf{c}$ have the type $\mathsf{u} \to A$ and G has the type $A \times A \to B$, x and y are first-order variables with the respective types $\mathsf{u} \to A$ and $\mathsf{u} \to B$.

Using second-order variables instead of first-order variables improves generalization because second-order variables abstract contexts. For example, $\mathsf{even}\ f\ \mathsf{fib}$ is a second-order generalization of $\mathsf{even\,fib}$ and $\mathsf{even\,S\,fib}$ by the substitutions $\{f \mapsto \mathsf{1}\}$ and $\{f \mapsto \mathsf{S}\}$. Thus in the first case f abstracts the empty context and in the second case f abstracts the context S. This allows the generalization to capture the common occurrence of fib. In contrast, first-order variables must generalize entire terms. For instance, the first-order generalization of $\mathsf{even\,fib}$ and $\mathsf{even\,S\,fib}$ is $\mathsf{even}\,x$, which loses the information that fib occurs in both terms. In the case of cartesian combinators, a second order generalization of $\mathsf{F}\langle \mathsf{a}, \mathsf{G}\langle \mathsf{a}, \mathsf{b}\rangle\rangle$ and $\mathsf{F}\langle \mathsf{c}, \mathsf{H}\langle \mathsf{b}, \mathsf{c}\rangle\rangle$ is $\mathsf{F}\langle x, f\langle x, \mathsf{b}\rangle\rangle$ by the substitutions $\{f \mapsto \mathsf{G}, x \mapsto \mathsf{a}\}$ and $\{f \mapsto \mathsf{H}\langle \pi', \pi\rangle, x \mapsto \mathsf{c}\}$. Again, the first-order generalization $\mathsf{F}\langle x, y\rangle$ is less useful because it fails to reflect the common b and the second occurrence of matching a to c. Thus using second-order variables improves generalization.

However, using second-order variables also introduces complexity. Because abstracting one context may preclude abstracting another, we get multiple maximally specific generalizations:

Example 5.3

$$\begin{tikzcd}
\mathsf{AB} & f_1\mathsf{A}f_2 \arrow[l, "{[\mathsf{1},\mathsf{B}]}"'] \arrow[r, "{[\mathsf{B},\mathsf{1}]}"] & \mathsf{BA} \\
\mathsf{AB} & g_1\mathsf{B}g_2 \arrow[l, "{[\mathsf{A},\mathsf{1}]}"'] \arrow[r, "{[\mathsf{1},\mathsf{A}]}"] & \mathsf{BA}
\end{tikzcd}$$

There is no substitution from both of these to some common generalization because substitutions cannot remove symbols. The existence of multiple maximally specific generalizations should not be surprising; it reflects the multiple ways of transforming one term into another. In this case, there are two ways to transform AB into BA: delete and append A or delete and prepend B. There is not enough information in AB and BA to be sure which sequence is intended. Which transformation sequence is more appropriate is best determined by examining other (possibly semantic) information.

Thus using second-order variables improves generalization but introduces complexity. In the first-order case, generalizations are unique so they can be defined by the greatest lower bound in a lattice. But in the second-order case, a different approach is needed. Our solution is given in Section 5.2. But first, we consider related work and define notation used throughout this chapter.

5.1.2 Related Work

Applications of higher-order logic has become an active area of research. In the early years of automated theorem proving, Robinson [Rob69] described a system of higher-order logic based on λ-calculus and showed how it would be useful for proving mathematical theorems. Since then, algorithms for higher-order unification have become available [Dar71, Hue75, HL78, PJ72, Pie73, Wol93], and these have lead to several systems based on λ-calculus such as those described in [AMCP84, FM88, Gor88, HM88, MN86, MCA82, Nip91, Pau86, PE88, Pfe88].

However, while higher-order unification has received much attention, higher-order generalization has not. Much of the work combining higher-order terms and generalization has been done in the context of explanation-based learning [DM86, MKKC86]. [Hag89] considers generalizing higher-order types from a single proof. [Hag90, Hag91b, Hag91a] show how to construct first-order programs from examples by unifying higher-order terms. [HJ92] uses a similar method to construct definite-clause grammars. However, all of this work uses an underlying theory to control generalization. For replay, such a theory is not available. That is, we are interested in similarity-based learning (*cf.* [Mic86])—*i.e*, learning from comparisons between examples—rather than explanation-based learning.

The work by Pfenning [Pfe91] is more directly related to the work in this chapter. Pfenning identifies a subset of generalizations which has useful properties. This subset consists of those generalizations which can be classified as *higher-order patterns*. Introduced in [MN87] and [Nip91], a λ-term (in β-normal form) is a higher-order pattern if every free occurrence of a variable f appears as $f(x_1, \ldots, x_n)$ where each x_i is a distinct, bound variable. For example, $\lambda xy.g(x)$ is a pattern, while $\lambda x.f(f(x))$, $\lambda x.f(x,x)$, and $f(\mathsf{a})$ are not. [MN87] shows that because maximally general unifiers between higher-order patterns are unique, they are very useful in higher-order logic. [Pfe91] shows that the substitution ordering on patterns gives a preorder with unique maximally specific generalizations for any pair of terms.[3] Thus the pattern restriction leads to well-defined generalizations.

Though higher-order patterns have many uses, they are not suitable for replay. The motivation for considering second-order terms is that second-order variables serve as contexts into which common subterms are embedded. With the pattern restriction, only bound variables can be embedded within contexts. Because bound variables do not play a major role in replaying program derivations, the result is that in most cases only first-order variables would appear within a generalization. Thus generalization using higher-order patterns is inadequate for replay for the same reason that first-order generalization is inadequate.

Other related work comes from generalization modulo a theory. Second-order generalization is a specific case of E-generalization; that is, generalization modulo an equational theory E. Again, while E-unification has received much attention (see [BS93] for a survey), E-generalization has not. Baader [Baa91] discusses generalization for commutative theories and gives a framework for general E-generalization. This framework is limited to systems in which there is at most one substitution between two solutions; a more general framework is needed for second-order generalization. Page and Frisch [FP90, PF92, Pag93] consider generalization with respect to taxonomic information such as "the mother of an elephant is an elephant." While their work is based on first-order logic, their framework is closely related to the framework presented in this chapter.

5.1.3 Notation

To emphasize composition, we sometimes write st as $s \circ t$. The set of free variables in the term t is denoted by $\mathcal{FV}(t)$. A term is *ground* if it contains no free variables. As mentioned in the first section, variables are denoted by *italics* and constants by $\mathsf{sans\ serif}$. Generally, we use lower case to denote first-order constants and upper case to denote second-order constants. The major exception is that when an example uses terms from the Focus programming language, we follow the convention of Focus in using upper case for constructor terms and lower case for defined functions. Which convention is being used will be obvious from the context.

A substitution θ is a finite map from variables to terms, $\{x \mapsto t, \ldots\}$. $dom(\theta)$ denotes the set of variables bound by θ and $ran(\theta)$ their bindings. $x \mapsto t$ is a *renaming* if t is a free variable y, and a substitution θ is a renaming if all $x \mapsto t \in \theta$ are renamings.[4] θ_{id} denotes the identity substitution (ambiguously for any set of variables). If applying θ to term s gives a term t, we variously denote this by $\theta(s)$, $\theta : s \to t$, or $s \xrightarrow{\theta} t$. Application of θ to s is defined only if $\mathcal{FV}(s) = dom(\theta)$. Dropping the $\theta :$ and writing just $s \to t$ asserts the existence of a substitution

[3] More precisely, these generalizations are unique modulo the names of the variables and permutations of the arguments of free variables.

[4] Note that $\{f \mapsto x, g \mapsto x\}$ is a renaming by this definition.

from s to t. $s \not\rightarrow t$ asserts that there is no substitution from s to t. Composition of substitutions $\sigma : r \rightarrow s$ and $\theta : s \rightarrow t$ is defined as

$$\theta \circ \sigma = \{x \mapsto \theta(\sigma(x)) \mid x \in dom(\sigma)\} : r \rightarrow t$$

Given a theory E, equality on substitutions is defined by

$$\theta =_E \sigma : s \rightarrow t \iff \forall x \in dom(\theta),\ \theta(x) =_E \sigma(x)$$

We usually leave E implicit. To make substitutions easier to read, we often omit the variables being bound. Specifically, we often write a substitution $\theta : s \rightarrow t$ as $[\theta(x_1), \ldots, \theta(x_n)]$ where $\langle x_1, \ldots, x_n \rangle$ is the sequence of free variables in s listed in the order they occur when reading from left to right. For example, $f(x, y, x) \xrightarrow{[a,b]} f(a, b, a)$.

5.2 Maximally Specific Second-Order Generalizations

Example 5.3 shows that a definition of second-order generalization should allow multiple solutions. This section presents such a definition motivated by similar properties of second-order unification. In particular, we discuss minimally complete sets of unifiers and define minimally complete sets of generalizations. We then show that such sets are not canonical, but that an additional condition gives sets which are. This results in generalizations which are useful for replay. An alternative motivation for our definitions is given in Appendix C which applies concepts from category theory to unification and generalization.

In the first-order case, generalization is defined by the greatest lower bound in the substitution ordering. However, Example 5.3 shows that greatest lower bounds do not exist for second-order terms. This is the same as in second-order unification: least upper bounds exist for first-order terms (if the terms are unifiable), but not for second-order terms:

Example 5.4

$$\begin{tikzcd} f_1\mathsf{A}f_2 \arrow[r, "{[x, y\mathsf{B}z]}"] & x\mathsf{A}y\mathsf{B}z & g_1\mathsf{B}g_2 \arrow[l, "{[x\mathsf{A}y, z]}"'] \\ f_1\mathsf{A}f_2 \arrow[r, "{[x\mathsf{B}y, z]}"] & x\mathsf{B}y\mathsf{A}z & g_1\mathsf{B}g_2 \arrow[l, "{[x, y\mathsf{A}z]}"'] \end{tikzcd}$$

There is no most general unifier of $f_1\mathsf{A}f_2$ and $g_1\mathsf{B}g_2$ because no unifier generalizes both $x\mathsf{A}y\mathsf{B}z$ and $x\mathsf{B}y\mathsf{A}z$.

Examples such as 5.4 lead to defining *sets* of maximally general unifiers [Plo72, BS93]. Let

$$\mathbf{U}(a, b) = \{\langle \theta_i : a \rightarrow t_i, \theta'_i : b \rightarrow t_i \rangle\}$$

denote the set of unifiers[5] of a and b, and let the application of a substitution to a unifier be defined by

$$\rho : \langle \theta : a \to t, \theta' : b \to t \rangle \to \langle \sigma : a \to s, \sigma' : b \to s \rangle \iff \rho \circ \theta = \sigma \text{ and } \rho \circ \theta' = \sigma'$$

We call such substitutions *unifier morphisms* or sometimes simply *morphisms*. Recall that we write $u_1 \to u_2$ to assert the existence of a morphism from u_1 to u_2. Then

Definition 5.5 Σ is a *minimally complete set of unifiers* (MCSU) of a and b if it satisfies each of the following properties:

Soundness: $\Sigma \subseteq \mathbf{U}(a, b)$.

Completeness: for all $u \in \mathbf{U}(a, b)$, there is a $v \in \Sigma$ such that $v \to u$.

Minimality: for all $u, v \in \Sigma$, $u \to v$ implies $u = v$.

For example, 5.4 gives an MCSU for $f_1 \mathsf{A} f_2$ and $g_1 \mathsf{B} g_2$.

These properties form the basis for defining second-order generalization as well. However, we first add another condition to the definition. Consider the following unifiers of x and y:

$$\begin{tikzcd}
u_1 : & x \arrow[r, "{[z]}"] & z & y \arrow[l, "{[z]}"'] \\
u_2 : & x \arrow[r, "{[xy]}"] & xy & y \arrow[l, "{[xy]}"'] \\
u_i : & x \arrow[r, "{[x_1 x_2 \ldots x_i]}"] & x_1 x_2 \ldots x_i & y \arrow[l, "{[x_1 x_2 \ldots x_i]}"']
\end{tikzcd}$$

Each set $\{u_i\}$ is an MCSU. Hence completeness and minimality do not lead to "canonical" sets of unifiers. The most obvious issue is that unifiers can have an arbitrary number of variables. But a second issue is that the sets are not equivalent in the sense that they are not isomorphic, where

Definition 5.6 Object a is *isomorphic* to b, written $a \cong b$, if there are morphisms f and g such that $f : a \to b$, $g : b \to a$, $g \circ f$ is the identity morphism on a, and $f \circ g$ is the identity morphism on b.

In the case of $\{u_1\}$ and $\{u_2\}$ we have

$$[xy] : u_1 \to u_2$$

[5]Usually unifiers are denoted by just substitutions; we denote them by pairs to make the source and destination of each substitution explicit and to allow a and b to share variables without confusion. These are often called *weak unifiers* [Ede85, Baa91].

$$[\mathbf{1}, z] : u_2 \rightarrow u_1$$

$$[\mathbf{1}, z] \circ [xy] = [z] = \theta_{id} : u_1 \rightarrow u_1$$

but

$$[xy] \circ [\mathbf{1}, z] = [\mathbf{1}, xy] \neq \theta_{id} : u_2 \rightarrow u_2$$

Using the alternative morphism $[z, \mathbf{1}] : u_2 \rightarrow u_1$ gives similar results. Thus $\{u_1\}$ and $\{u_2\}$ are not isomorphic. In fact, $\{u_i\} \ncong \{u_j\}$ for all distinct i and j.

For many applications, having a canonical MCSU is unimportant. For instance, a theorem prover can choose any MCSU since the primary concern is completeness. But in other applications, multiple MCSUs may be undesirable. This particularly true for logic programming languages in which the unification algorithm determines the computed answer sets. If there is no canonical solution, different implementations may produce different answer sets.

One solution is to choose the MCSU with the fewest variables in each unifier. However, it is not obvious which variables should be eliminated. An alternative approach is to add a uniqueness condition:

Uniqueness: If $\rho : u \rightarrow v$ and $\rho' : u \rightarrow v$ for $u \in \mathbf{U}(a, b)$ and $v \in \Sigma$, then $\rho = \rho'$.

This condition is motivated by discussions in [Gog89]; see Appendix C. In the above example, only $\{u_1\}$ satisfies uniqueness; $\{u_i\}$ for $i > 1$ does not. Thus, at least in this example, uniqueness serves to restrict the number of variables. It also results in MCSUs which are canonical up to an isomorphism:

Theorem 5.7 If Σ_1 and Σ_2 are minimally complete sets of unifiers of a and b satisfying the uniqueness condition, $\Sigma_1 \cong \Sigma_2$.

Proof By assumption, for each $s_1 \in \Sigma_1$ there is an $s_2 \in \Sigma_2$ and a unique morphism ρ such that $\rho : s_1 \rightarrow s_2$. Likewise, there is an $s_1' \in \Sigma_1$ and a unique morphism $\rho' : s_2 \rightarrow s_1'$. But because substitutions compose and Σ_1 satisfies the minimality condition, $s_1 = s_1'$. Furthermore, the only morphism from s_1 to itself is the identity substitution, so $\rho' \circ \rho = \theta_{id} : s_1 \rightarrow s_1$. Likewise, $\rho \circ \rho' = \theta_{id}$, thus $\Sigma_1 \cong \Sigma_2$. $\square$

The same issues arise in generalization. The dual of Definition 5.5 can be used to define generalization, but completeness and minimality do not lead to canonical sets. Consider the generalizations

Example 5.8

$$\begin{tikzcd} g_1 : & \mathsf{A} & x'z' \arrow[l, "{[\mathsf{A}, \mathbf{1}]}"'] \arrow[r, "{[\mathbf{1}, \mathsf{B}]}"] & \mathsf{B} \end{tikzcd}$$

$$\begin{tikzcd} g_2 : & \mathsf{A} & xyz \arrow[l, "{[\mathsf{A}, \mathbf{1}, \mathbf{1}]}"'] \arrow[r, "{[\mathbf{1}, \mathbf{1}, \mathsf{B}]}"] & \mathsf{B} \end{tikzcd}$$

Applying $[x', \mathbf{1}, z']$ to g_2 gives g_1 and applying either $[xy, z]$ or $[x, yz]$ to g_1 gives g_2, but $g_1 \not\cong g_2$. In general, there may be an arbitrary number of variables in generalizations, most of which are bound to $\mathbf{1}$.

As for unification, we obtain canonical sets of generalizations by introducing a uniqueness condition. This has the result of eliminating extra variables such as the y in g_2. Variables in generalizations mark differences between terms, so they should play non-trivial roles in any results examined by the user. In Theorem 5.16 below, we show that introducing uniqueness eliminates such trivial variables.

Dualizing unification gives the following definition of generalization: Let

$$\mathbf{G}(a, b) = \{\langle \theta_i : t_i \to a, \theta'_i : t'_i \to b \rangle\}$$

denote the set of generalizations of a and b, and let the application of a substitution to a generalization be defined by

$$\begin{aligned} \rho : \langle \theta : t \to a, \theta' : t \to b \rangle &\to \langle \sigma : s \to a, \sigma' : s \to b \rangle \\ &\iff \rho(t) = s,\ \theta = \rho \circ \sigma,\ \text{and } \theta' = \rho \circ \sigma' \end{aligned}$$

We illustrate this by a diagram:[6]

$$\begin{tikzcd} & t \arrow[dl, "\theta"'] \arrow[d, "\rho"] \arrow[dr, "\theta'"] & \\ a & s \arrow[l, "\sigma"'] \arrow[r, "\sigma'"] & b \end{tikzcd}$$

We say that such a diagram *commutes* if for any two paths between two terms, the compositions of the substitutions along each path give the same substitution. Thus the conditions $\theta = \rho \circ \sigma$ and $\theta' = \rho \circ \sigma'$ are satisfied if and only if the above diagram commutes.

Definition 5.9 Γ is a *set of maximally specific generalizations* of a and b if

Soundness: $\Gamma \subseteq \mathbf{G}(a, b)$.

Completeness: For all $g \in \mathbf{G}(a, b)$, there is a $g' \in \Gamma$ such that $g \to g'$.

Minimality: For all $g, g' \in \Gamma$, $g \to g'$ implies $g = g'$.

Uniqueness: If $\rho : g \to g'$ and $\rho' : g \to g'$ for $g \in \mathbf{G}(a, b)$ and $g' \in \Gamma$, then $\rho = \rho'$.

[6]This and later diagrams in this thesis were formatted using Paul Taylor's commutative diagrams style package.

We call substitutions between generalizations *generalization morphisms*. Substituting Γ for Σ in the proof of Theorem 5.7 gives

Theorem 5.10 If Γ_1 and Γ_2 are maximally specific generalizations of a and b, $\Gamma_1 \cong \Gamma_2$.

Proof Dual of Theorem 5.7. $\square$

Since maximally specific generalizations are unique up to an isomorphism, we often refer to *the* set of maximally specific generalizations even though there is usually more than one. We denote this set by MSG(a, b).

Example 5.11 The generalizations in Example 5.3,

$$\begin{tikzcd}\mathsf{AB} & f_1\mathsf{A}f_2 \arrow[l, "{[\mathbf{1},\mathsf{B}]}"'] \arrow[r, "{[\mathsf{B},\mathbf{1}]}"] & \mathsf{BA}\end{tikzcd}$$

$$\begin{tikzcd}\mathsf{AB} & g_1\mathsf{B}g_2 \arrow[l, "{[\mathsf{A},\mathbf{1}]}"'] \arrow[r, "{[\mathbf{1},\mathsf{A}]}"] & \mathsf{BA}\end{tikzcd}$$

comprise MSG$(\mathsf{AB}, \mathsf{BA})$. This is because the only other generalizations in $\mathbf{G}(\mathsf{AB}, \mathsf{BA})$ containing A (for example) have morphisms to generalizations with only two variables. This is illustrated by the commutative diagram

$$\begin{tikzcd} & f_1 \ldots f_n\mathsf{A}g_1 \ldots g_m \arrow[dl, "\theta_1"'] \arrow[d, "\rho"] \arrow[dr, "\theta_2"] & \\ \mathsf{AB} & f\mathsf{A}g \arrow[l, "{[\mathbf{1},\mathsf{B}]}"'] \arrow[r, "{[\mathsf{B},\mathbf{1}]}"] & \mathsf{BA}\end{tikzcd}$$

in which θ_1, θ_2, and ρ map all variables to $\mathbf{1}$ except for some $f_k \in \{f_1 \ldots f_n\}$ and some $g_{k'} \in \{g_1 \ldots g_n\}$ such that $f_k \mapsto \mathsf{B} \in \theta_1$, $g_{k'} \mapsto \mathsf{B} \in \theta_2$, and $\{f_k \mapsto f, g_{k'} \mapsto g\} \subseteq \rho$.

We use the following result to establish cases for which MSG is defined:

Theorem 5.12 MSG(a, b) exists if there is a subset $\mathcal{G}$ of $\mathbf{G}(a, b)$ having the following properties:

i. $\mathcal{G}$ is finite,

ii. for each $g \in \mathbf{G}(a, b)$, there is a $g' \in \mathcal{G}$ such that $g \to g'$, and

iii. if $g \in \mathbf{G}(a, b)$, $g' \in \mathcal{G}$, $\rho : g \to g'$, and $\rho' : g \to g'$, then $\rho = \rho'$.

This follows immediately from Definition 5.9.

The next three sections consider the existence of maximally specific generalizations for different classes of terms. We first examine the monadic combinators because they are relatively simple to understand and because they illustrate most of the issues. Following this, we consider the extension to multiple arguments using cartesian combinators and show that maximally specific generalizations do not exist for this class of terms. This leads to a representation for multiple arguments for which maximally specific generalizations do exist.

5.3 Generalization of Monadic Combinator Terms

This section considers generalizing terms over (typed) monadic function symbols. We give a formal definition of the monadic combinator terms, show that matching such terms is decidable, and that maximally specific generalizations exist. We then show that the full set contains too many generalizations and consider a restriction that gives a more useful set of generalizations. Finally, we give a semi-efficient algorithm for computing the restricted generalizations for ground terms.

Let $\mathcal{C}_{A\to B}$ be the set of constants of type $A \to B$, and let $\mathcal{V}_{A\to B}$ be the set of variables of type $A \to B$. Then the set of monadic combinator terms of type $A \to B$, $\mathcal{T}_{A\to B}$, is the least set of terms consistent with the rules

$$\frac{c \in \mathcal{C}_{X\to Y}}{c : X \to Y} \qquad \frac{f \in \mathcal{V}_{X\to Y}}{f : X \to Y} \qquad \frac{}{\mathbf{1}_X : X \to X} \qquad \frac{s : Y \to Z \quad t : X \to Y}{s\,t : X \to Z} \tag{5.4}$$

We assume the following equivalences among terms:

$$\mathbf{1}_B\, t = t \qquad t\,\mathbf{1}_A = t \qquad (rs)t = r(st) \tag{5.5}$$

Formally, a "term" is an equivalence class of terms which is the least congruence relation generated by these equations.[7] We assume that the sets $\mathcal{C}_{A\to B}$ and $\mathcal{V}_{A\to B}$ are pairwise disjoint. By this assumption, the constants and variables determine the type of a term, allowing us to omit type information.

The next sections examine matching and generalizing monadic combinator terms.

5.3.1 Matching

Given terms r and s, matching is the process of finding a maximally general substitution θ such that $\theta : r \to s$. We discuss matching to improve our understanding of monadic combinator terms, to establish its decidability, and because matching plays a key role in the definition of MSG.

[7]In fact, "terms" are simply the arrows of the free category (with types as objects) generated by constants $\mathcal{C}_{X\to Y}$ and variables $\mathcal{V}_{X\to Y}$.

Matching monadic combinators is a form of string matching [CLR90] and so is decidable. In particular, we can find a solution by matching terms from left to right, substituting some prefix of the unmatched portion (including the empty string) for variables as they are encountered. This is constrained by the requirement that substitutions respect the types of the variables.

Example 5.13 The following sequence illustrates finding the θ such that $f\mathsf{CC}gf\mathsf{C} \rightarrow \mathsf{ABCCABC}$. Assume all of the constants and variables have the type $X \rightarrow X$.

$$\begin{array}{rll}
f\mathsf{CC}gf\mathsf{C} & \mathsf{ABCCABC} & \text{assume } f \mapsto \mathsf{AB} \in \theta \\
\mathsf{CC}g\mathsf{ABC} & \mathsf{CCABC} & \\
g\mathsf{ABC} & \mathsf{ABC} & \text{assume } g \mapsto \mathbf{1} \in \theta \\
\mathsf{ABC} & \mathsf{ABC} & \\
\mathbf{1} & \mathbf{1} &
\end{array}$$

Thus $[\mathsf{AB}, \mathbf{1}] : f\mathsf{CC}gf\mathsf{C} \rightarrow \mathsf{ABCCABC}$. This is the only substitution between these two terms.

In general, there may be more than one substitution:

$$[\mathsf{A}, \mathsf{B}], [\mathbf{1}, \mathsf{AB}], [\mathsf{AB}, \mathbf{1}] : \ fg \rightarrow \mathsf{AB}$$

The following terms cannot be matched:

$$\begin{array}{rcl}
f\mathsf{A} & \not\rightarrow & \mathsf{B} \\
f\mathsf{AA} & \not\rightarrow & \mathsf{A}
\end{array}$$

In the first case, there is no way to convert an A to a B, and in the second case there is no way to delete an A.

5.3.2 Generalization

We now turn to examining generalization over monadic combinator terms. We show that maximally specific generalizations exist for any pair of monadic combinator terms having the same type.

More specifically, we show that $\mathrm{MSG}(a, b)$ is defined for the following subset of $\mathbf{G}(a, b)$:

Definition 5.14 Let $\mathbf{G}'(a, b)$ be the subset of $\mathbf{G}(a, b)$ such that if $\langle \theta_1 : s \rightarrow a, \theta_2 : s \rightarrow b \rangle \in \mathbf{G}'(a, b)$ and $f \in \mathcal{FV}(s)$, then $\theta_1(f) = \theta_2(f)$ implies $\theta_1(f) = f$.

Because constants cannot be eliminated by substitution, the number of constants and variables in generalizations in $\mathbf{G}'(a, b)$ is bounded by $|a|$ and $|b|$, hence

Lemma 5.15 $\mathbf{G}'(a, b)$ is finite.

Theorem 5.16 MSG(a, b) exists.

Proof We show that there is a subset of $\mathbf{G}(a, b)$ satisfying the conditions of Theorem 5.12 by showing that morphisms to generalizations in $\mathbf{G}'(a, b)$ are unique. That is, if g_1 is $\langle \theta_1 : s \to a, \theta_2 : s \to b \rangle$ in $\mathbf{G}(a, b)$, g_2 is $\langle \sigma_1 : t \to a, \sigma_2 : t \to b \rangle$ in $\mathbf{G}'(a, b)$, and $\rho, \rho' : g_1 \to g_2$ in $\mathbf{G}(a, b)$, then we must show that $\rho = \rho'$ in the picture

$$\begin{tikzcd}
 & s \arrow[dl, "\theta_1"'] \arrow[d, shift right, "\rho"'] \arrow[d, shift left, "\rho'"] \arrow[dr, "\theta_2"] & \\
a & t \arrow[l, "\sigma_1"'] \arrow[r, "\sigma_2"] & b
\end{tikzcd}$$

Example 5.8 shows that this condition is not guaranteed if $g_2 \notin \mathbf{G}'(a, b)$.

We use proof by contradiction. Let s be rfr' for some r and r' where f is the leftmost variable in s such that $\rho(f) \neq \rho'(f)$. Also, let $\rho(f) = pq$ and $\rho'(f) = pq'$ such that p is the largest common prefix of $\rho(f)$ and $\rho'(f)$. Then $t = \rho(r) \circ pq \circ \rho(r') = \rho'(r) \circ pq' \circ \rho'(r')$, and since $\rho(r) = \rho'(r)$, $q \circ \rho(r') = q' \circ \rho'(r')$. $q \neq q'$, so either q or q' must be $\mathbf{1}$; assume it is q. Because $\sigma_1 \circ \rho = \sigma_1 \circ \rho'$, $\sigma_1(q') = \mathbf{1}$, so $q' = hq''$ where h is a variable such that $\sigma_1(h) = \mathbf{1}$. Likewise, $\sigma_2(h) = \mathbf{1}$. But this contradicts g_2 in $\mathbf{G}'(a, b)$, so $\rho = \rho'$. □

Example 5.17 MSG$(\mathsf{FA}, \mathsf{GA})$ is

$$\begin{tikzcd}
\mathsf{FA} & h_1h_2\mathsf{A} \arrow[l, "{[\mathsf{F}, \mathbf{1}]}"'] \arrow[r, "{[\mathbf{1}, \mathsf{G}]}"] & \mathsf{GA}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathsf{FA} & h_1h_2\mathsf{A} \arrow[l, "{[\mathbf{1}, \mathsf{F}]}"'] \arrow[r, "{[\mathsf{G}, \mathbf{1}]}"] & \mathsf{GA}
\end{tikzcd}$$

This shows that second-order generalization captures common subterms. These generalizations are maximally specific because any generalization with more variables would not be in $\mathbf{G}'(\mathsf{FA}, \mathsf{GA})$.

Thus maximally specific generalizations exist for monadic combinator terms. However, Example 5.17 suggests that such generalizations are not as useful as they might be. The next section examines this problem and gives a solution.

5.3.3 Restricting MSG

While MSG over monadic combinator terms is well-defined, Example 5.17 suggests that more restrictions are needed. The transforms from this example are

$$\mathsf{F} \Rightarrow \mathbf{1}$$
$$\mathbf{1} \Rightarrow \mathsf{G}$$

These split the transformation from F to G into two parts: deleting F and introducing G. The more natural generalization of FA and GA is

Example 5.18

$$\begin{tikzcd}\mathsf{FA} & f\mathsf{A} \arrow[l, "{[\mathsf{F}]}"'] \arrow[r, "{[\mathsf{G}]}"] & \mathsf{GA}\end{tikzcd}$$

which captures that F has been replaced by G. This section considers identifies a restricted set of maximally specific generalization, illustrates that it contains natural generalizations, and considers its effect on generalization.

There are several problems with the generalizations in Example 5.17. First, Example 5.18 shows that it is not the generalization we would expect. Furthermore, since any subterm contains an instance of $\mathbf{1}$, the transform $\mathbf{1} \Rightarrow \mathsf{G}$ is not effective because it can be applied anywhere. A third problem is that as the terms grow larger, we obtain even more generalizations. MSG(AB, CD) contains six generalizations of the form

$$\begin{tikzcd}\mathsf{AB} & f_1 f_2 f_3 f_4 \arrow[l, "{[\mathsf{A}, \mathsf{B}, \mathbf{1}, \mathbf{1}]}"'] \arrow[r, "{[\mathbf{1}, \mathbf{1}, \mathsf{C}, \mathsf{D}]}"] & \mathsf{CD}\end{tikzcd}$$

and

$$\begin{tikzcd}\mathsf{AB} & f_1 f_2 f_3 f_4 \arrow[l, "{[\mathsf{A}, \mathbf{1}, \mathsf{B}, \mathbf{1}]}"'] \arrow[r, "{[\mathbf{1}, \mathsf{C}, \mathbf{1}, \mathsf{D}]}"] & \mathsf{CD}\end{tikzcd}$$

But perhaps the most serious problem is that adding terms (such as generalizing EF as well as AB and CD) does little except increase the number of generalizations and the size of generalization terms.

The problem is caused by the presence of useless variables in the generalization term. For instance, h_1 in Example 5.17 plays an unimportant role in the substitution $[\mathbf{1}, \mathsf{G}] : h_1 h_2 \mathsf{A} \rightarrow \mathsf{GA}$. The only reason for not using $[\mathsf{G}, \mathbf{1}]$ instead is that then the substitutions for h_2 would be identical and so h_2 could be removed. Our solution is to disallow variables that are "redundant" with other variables in the sense that the generalization can be formed without the extra variables:

Definition 5.19 A pair of free variables f and g in term t are *adjacent* if t contains a subterm equal to fg.

Definition 5.20 A generalization $\langle \theta_1 : t \rightarrow a_1, \theta_2 : t \rightarrow a_2 \rangle$ is *redundant* if f and g are adjacent free variables in t and any of the following is true: $\theta_1(f) \neq f$, $\theta_2(f) \neq f$, $\theta_1(g) \neq g$, or $\theta_2(g) \neq g$.

Thus the generalizations in Example 5.17 are redundant because h_1 and h_2 are adjacent and (for instance) $\theta_1(h_1) = \mathsf{F}$.

Definition 5.21 We define $\mathbf{CG}(a,b)$, the *condensed* generalizations, to be the subset of $\mathbf{G}(a,b)$ not containing redundant generalizations. The maximally specific condensed generalizations, denoted by $\mathrm{MSC}(a,b)$, is the set of maximally specific generalizations in $\mathbf{CG}(a,b)$.

Since $\mathbf{CG}(a,b)$ is a subset of $\mathbf{G}(a,b)$,

Theorem 5.22 $\mathrm{MSC}(a,b)$ exists for any monadic combinator terms a,b having the same type.

Examples 5.23 The following examples illustrate MSC:

- $\mathrm{MSC}(\mathsf{FA},\mathsf{GA}) = \langle[\mathsf{F}] : f\mathsf{A} \to \mathsf{FA}, [\mathsf{G}] : f\mathsf{A} \to \mathsf{GA}\rangle$, as desired.
- $\mathrm{MSC}(\mathsf{AB},\mathsf{CD}) = \{[\mathsf{AB}] : f \to \mathsf{AB}, [\mathsf{CD}] : f \to \mathsf{CD}\}$; all other generalizations contain redundant variables.
- $\mathrm{MSC}(\mathsf{A},\mathsf{A}) = \{\theta_{id} : \mathsf{A} \to \mathsf{A}, \theta_{id} : \mathsf{A} \to \mathsf{A}\}$. In contrast, $\mathrm{MSG}(\mathsf{A},\mathsf{A})$ includes generalizations such as $\langle[\mathbf{1},\mathsf{A}] : fg \to \mathsf{A}, [\mathsf{A},\mathbf{1}] : fg \to \mathsf{A}\rangle$. Thus MSC contains more natural generalizations in even simple cases.
- As with MSG, MSC may contain more than one generalization:
  $\mathrm{MSC}(\mathsf{AB},\mathsf{BA}) =$

$$\begin{tikzcd}
\mathsf{AB} & f_1\mathsf{A}f_2 \arrow[l, "{[\mathbf{1},\mathsf{B}]}"'] \arrow[r, "{[\mathsf{B},\mathbf{1}]}"] & \mathsf{BA}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathsf{AB} & g_1\mathsf{B}g_2 \arrow[l, "{[\mathsf{A},\mathbf{1}]}"'] \arrow[r, "{[\mathbf{1},\mathsf{A}]}"] & \mathsf{BA}
\end{tikzcd}$$

  Likewise, $\mathrm{MSC}(\mathsf{BB},\mathsf{ABC}) =$

$$\begin{tikzcd}
\mathsf{BB} & f_1\mathsf{B}f_2 \arrow[l, "{[\mathsf{B},\mathbf{1}]}"'] \arrow[r, "{[\mathsf{A},\mathsf{C}]}"] & \mathsf{ABC}
\end{tikzcd}$$

$$\begin{tikzcd}
\mathsf{BB} & g_1\mathsf{B}g_2 \arrow[l, "{[\mathbf{1},\mathsf{B}]}"'] \arrow[r, "{[\mathsf{A},\mathsf{C}]}"] & \mathsf{ABC}
\end{tikzcd}$$

  The second case shows that multiple generalizations can arise from multiple substitutions between terms.

- Example 4.2 (p. 38) provides an example from Focus: MSC(len fringe tree, size tree) is

$$\begin{tikzcd}
\textsf{len fringe tree} & f\ \textsf{tree} \arrow[l, "{[\textsf{len fringe}]}"'] \arrow[r, "{[\textsf{size}]}"] & \textsf{size tree}
\end{tikzcd}$$

  In contrast, [Pfe91] disallows this generalization because f tree is not a higher-order pattern. The only generalization satisfying the pattern restriction is

$$\begin{tikzcd}
\textsf{len fringe tree} & x \arrow[l, "{[\textsf{len fringe tree}]}"'] \arrow[r, "{[\textsf{size tree}]}"] & \textsf{size tree}
\end{tikzcd}$$

This generalization is not useful because the rule obtained from it cannot be used to transform the lemma

$$\mathsf{len(fringe(t))} + \mathsf{u} == 0 \rightarrow \mathsf{False}$$

This illustrates why higher-order patterns are not appropriate for replay.

As it turns out, morphisms between generalizations in $\mathbf{CG}(a, b)$ are unique; that is, $\mathbf{CG}(a, b)$ is a preorder (for the basis of a proof, see [HR92]). This allows us to introduce the following notation:

Definition 5.24 Whenever $g_1 \rightarrow g_2$ in $\mathbf{CG}(a, b)$, we say that g_1 is *more general* (or, equivalently, *less specific*) than g_2. This is written as $g_1 \geq g_2$. We write $g_1 > g_2$ if $g_2 \not\rightarrow g_1$ as well.

Alternative definitions, however, do not result in a preorder. For example, we could allow adjacent variables f and g in $\langle \theta_1 : t \rightarrow a_1, \theta_2 : t \rightarrow a_2 \rangle$ if neither θ_1 nor θ_2 maps f or g to $\mathbf{1}$. This represents a compromise between the minimal number of variables in MSC and their proliferation in MSG. This looser restriction leads to a system in which generalization morphisms are not always unique, as illustrated by the following example:[8]

Example 5.25

$$\begin{tikzcd}
 & f_1 f_2 f_3 \arrow[dl, "{[\mathsf{A}, \mathsf{B}, \mathsf{C}]}"'] \arrow[d, shift right, "{[\mathsf{A}f, \mathsf{B}, g]}"'] \arrow[d, shift left, "{[\mathsf{A}, f\mathsf{B}, g]}"] \arrow[dr, "{[\mathsf{A}, \mathsf{B}, \mathsf{D}]}"] & \\
\mathsf{ABC} & \mathsf{A}f\mathsf{B}g \arrow[l, "{[\mathbf{1}, \mathsf{C}]}"'] \arrow[r, "{[\mathbf{1}, \mathsf{D}]}"] & \mathsf{ABD}
\end{tikzcd}$$

Which definition of redundance works best depends upon the application;[9] we consider only the most restrictive version here because it works well for replay when we make application explicit and because the algorithm to compute it is more natural. Exploring other definitions of redundance is left as future work.

In this section, we have identified a restricted set of maximally specific generalizations, MSC, and gave examples showing that this set provides more natural generalizations. In the next section, we show how to compute this set.

5.3.4 Computing MSC

Since $\mathbf{G}'(a, b)$ (and so $\mathbf{CG}(a, b)$) is finite and matching monadic combinator terms is decidable, we can find MSC(a, b) by generating $\mathbf{CG}(a, b)$ and comparing all its objects against one an-

[8] Note that the bottom generalization is not in $\mathbf{G}'(\mathtt{ABC}, \mathtt{ABD})$, so this example does not contradict Theorem 5.16.

[9] This is why we did not disallow adjacent variables to obtain canonical sets of generalizations in Section 5.2.

other. However, there is a more practical algorithm suggested by the observation that given a generalization $\langle \theta_1 : t \to a, \theta_2 : t \to b \rangle$ in which some substitution in θ_1 and θ_2 contains a common subterm, then the subterm can be factored out. This section presents this algorithm and proves its correctness.

The algorithm presented here is restricted to generalizing ground terms. Extending it to non-ground terms is possible, but adds complexity, particularly in the correctness proof. It is also not necessary for our application: generalizing programs in replay does not involve non-ground terms. Thus the algorithm is restricted to ground terms for simplicity.

While this algorithm is more practical than generating all of $\mathbf{CG}(a,b)$ and searching for maximally specific generalizations, it is still expensive because it computes the same generalization in multiple ways. But it is useful to examine the algorithm because it characterizes the maximally specific condensed generalizations. Furthermore, this algorithm is also useful for combinators supporting multiple arguments. For methods that lead to a much more efficient, though less intuitive, generalization algorithm, see [Mye86]. [MM85] describes using the more efficient algorithm to implement the popular file-comparison program `diff`. Thus a third contribution from the following algorithm is that it relates generalization and the `diff` utility.

The steps for specializing generalizations of a and b are given by the following rules. The rules maintain the invariant that if generalization g_1 in $\mathbf{CG}(a,b)$ is transformed into g_2, then g_2 is a generalization in $\mathbf{CG}(a,b)$ and $g_1 > g_2$. First, note that

Observation 5.26 If $g \in \mathbf{CG}(a,b)$, then $g_0 = \langle [a] : x \to a, [b] : x \to b \rangle \geq g$.

This observation suggests a starting point for computing MSC.

Algorithm 5.27 To compute MSC(a,b), we start with $g_0 \in \mathbf{CG}(a,b)$ and continue specializing the generalization until no rule is applicable. To simplify the notation, we represent each generalization $\langle \theta_1 : t \to a,\ \theta_2 : t \to b \rangle$ by the triple $\langle t,\ \theta_1,\ \theta_2 \rangle$. The rules are of the form

$$\frac{t,\ \theta_1,\ \theta_2}{t',\ \theta_1',\ \theta_2'}$$

where $\langle t', \theta_1', \theta_2' \rangle$ is (strictly) more specific than $\langle t, \theta_1, \theta_2 \rangle$.

Delete Variables with the same binding in both substitutions can be removed:

$$\frac{t,\ \theta_1 \cup \{f \mapsto s\},\ \theta_2 \cup \{f \mapsto s\}}{\{f \mapsto s\}(t),\ \theta_1,\ \theta_2}$$

Merge Variables with the same bindings within each substitution can be merged:

$$\frac{t,\ \theta_1 \cup \{f \mapsto r, f' \mapsto r\},\ \theta_2 \cup \{f \mapsto s, f' \mapsto s\}}{\{f' \mapsto f\}(t),\ \theta_1 \cup \{f \mapsto r\},\ \theta_2 \cup \{f \mapsto s\}}$$

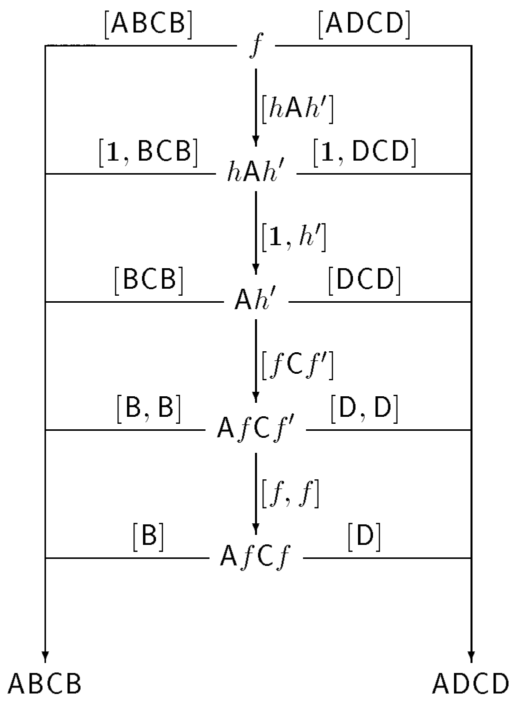


Figure 5.1: Generalizing ABCB and ADCD.

Factor Finally, common constants can be factored out:

$$\frac{t,\ \theta_1 \cup \{f \mapsto rKr'\},\ \theta_2 \cup \{f \mapsto sKs'\}}{\{f \mapsto hKh'\}(t),\ \theta_1 \cup \{h \mapsto r, h' \mapsto r'\},\ \theta_2 \cup \{h \mapsto s, h' \mapsto s'\}}$$

where $K \in \mathcal{C}_{X \to Y}$ and where h and h' are fresh variables.

We use $g_1 \longrightarrow g_2$ to denote applying a step to transform g_1 into g_2. The transitive closure of $\longrightarrow$ is written as $\longrightarrow^*$. Using $\longrightarrow$, the set MSC can be computed by *gen* defined as

$$gen(a, b) \quad = \quad \{g \mid g_0 \longrightarrow^* g, \text{ and } \nexists g' \text{ such that } g \longrightarrow g'\}$$

Example 5.28 Figure 5.1 illustrates each of the steps of Algorithm 5.27. From top to bottom, the steps are **Factor, Delete, Factor,** and **Merge**. There is one other sequence in which C is factored out before A, but the final result is the same.

$$\begin{tikzcd}
& t \arrow[ddl, "\theta_1"'] \arrow[d, "\rho_t"'] \arrow[dd, bend left, "\rho_r"] \arrow[ddr, "\theta_2"] & \\
& s \arrow[dl, "\sigma_1"] \arrow[d, "\rho_s"'] \arrow[dr, "\sigma_2"'] & \\
a & r \arrow[l, "\phi_1"'] \arrow[r, "\phi_2"] & b
\end{tikzcd}$$

Figure 5.2: Construction for Lemma 5.31.

5.3.4.1 Correctness

The remainder of this section shows that $gen(a,b)$ computes MSC(a,b). The heart of the proof is showing that the $\longrightarrow$-rules completely specify when one generalization is strictly more specific than another. The reader may skip ahead to Section 5.4 without missing crucial material.

Lemma 5.29 If g_1 is in $\mathbf{CG}(a,b)$ and $g_1 \longrightarrow g_2$, then g_2 is in $\mathbf{CG}(a,b)$.

Lemma 5.30 Whenever g_1, g_2 are in $\mathbf{CG}(a,b)$ and $g_1 \longrightarrow g_2$, $g_1 > g_2$.

Proof Observe that each step is of the form $\langle t, \theta_1, \theta_2\rangle \longrightarrow \langle \rho(t), \theta_1', \theta_2'\rangle$ where ρ is a generalization morphism. This gives $g_1 \geq g_2$. To complete the proof, we must show $g_2 \not\geq g_1$. This is trivial for the **Delete** and **Merge** steps. For the **Factor** step, $g_2 \not\geq g_1$ because there is no substitution which deletes a constant. □

Finally, we show that $\longrightarrow$-steps do not reduce the number of possible generalizations. That is, given a specific generalization, the set of $\longrightarrow$-steps completely covers all maximal generalizations which are more instantiated than the given one.

Lemma 5.31 Whenever $g_r \in$ MSC(a,b), g_t is in $\mathbf{CG}(a,b)$, and $g_t > g_r$, there is a g_s in $\mathbf{CG}(a,b)$ such that $g_t \longrightarrow g_s$ and $g_s \geq g_r$.

Proof Let g_t be $\langle \theta_1 : t \rightarrow a,\ \theta_2 : t \rightarrow b\rangle$, g_s be $\langle \sigma_1 : s \rightarrow a,\ \sigma_2 : s \rightarrow b\rangle$, and g_r be $\langle \phi_1 : r \rightarrow a,\ \phi_2 : r \rightarrow b\rangle$. Then Figure 5.2 illustrates the relationships between g_t, g_s, and g_r: We show that for any ρ_r, there is a $\longrightarrow$-step which generates an appropriate g_s. In each case, we only identify which step is applicable; refer to the algorithm for the details of constructing g_s and ρ_s.

If all substitutions in ρ_r are renamings, then there are $f, f' \in \mathcal{FV}(t)$ such that $f \neq f'$ and $f \mapsto f' \in \rho_r$. Thus **Merge** can be applied. Otherwise, choose $f \in \mathcal{FV}(t)$ such that $\rho_r(f)$ is not a renaming, and let $\rho_r(f) = Hp$ such that $H \neq \mathbf{1}$ unless $p = \mathbf{1}$. There are three cases for H:

i. $H = \mathbf{1}$: $\theta_1(f) = \phi_1(\rho_r(f)) = \mathbf{1} = \phi_2(\rho_r(f)) = \theta_2(f)$, so **Delete** can be applied.

ii. H is a constant K ($\neq \mathbf{1}$): $\mathit{head}(\theta_1(f)) = \mathit{head}(\phi_1(\rho_r(f))) = K = \mathit{head}(\phi_2(\rho_r(f))) = \mathit{head}(\theta_2(f))$ where head is defined as

$$\mathit{head}(t) = \begin{cases} \mathbf{1} & \text{if } t = \mathbf{1} \\ x & \text{if } t = xs \text{ for a symbol } x \neq \mathbf{1} \end{cases}$$

Thus **Factor** can be applied.

iii. H is a free variable: By assumption, $p \neq \mathbf{1}$. If $\mathit{head}(p)$ is a free variable, then g_r is redundant, a contradiction. If $\mathit{head}(p)$ is a constant, then $\mathit{head}(p)$ must appear in both $\theta_1(f)$ and $\theta_2(f)$, so **Factor** can be applied. □

These three lemmas give us

Theorem 5.32 (Soundness) If g is in $\mathit{gen}(a, b)$, then there is a $g' \in \mathrm{MSC}(a, b)$ such that g is isomorphic to g'.

and

Theorem 5.33 (Completeness) If g is in $\mathrm{MSC}(a, b)$, then there is a $g' \in \mathit{gen}(a, b)$ such that g is isomorphic to g'.

Thus MSC over monadic combinator terms is well-defined, useful, and computable. We next consider extensions to support pairs of terms.

5.4 Generalization of Combinators for Product Types

While generalizing monadic combinator terms provides useful results, support is needed for functions with multiple arguments. As described in Section 5.1.1, the standard representation for multiple arguments is to use cartesian products. In this section, we define the *cartesian combinator terms* and show that while matching cartesian combinator terms is well-defined, generalization is not. This leads to considering another representation of pairs of terms for which generalization is well-defined.

To incorporate products, we extend $\mathcal{T}_{A \to B}$ by adding the following rules to the rules (5.4) for monadic combinators:[10]

$$\frac{s : X \to Y \qquad t : X \to Z}{\langle s, t\rangle : X \to Y \times Z} \qquad \frac{}{\pi_{X,Y} : X \times Y \to X} \qquad \frac{}{\pi'_{X,Y} : X \times Y \to Y}$$

$$\frac{}{!_X : X \to \mathsf{u}}$$

The first row introduces products and functions to access their components as described in Section 5.1.1. The second row introduces the function !. This is the function which sends any type to the unit type, u. As described in the Section 5.1.1, the unit type is used to represent closed terms. $!_X$ is used to ignore an input by mapping it to the unit type. In λ-calculus, $t!$ would be written as $\lambda x.t$ where x does not occur in t. To maintain the distinction between ! and other combinators, we assume that there are no functions of type $X \to \mathsf{u}$ except $!_X$ (note that $\mathbf{1}_{\mathsf{u}} = !_{\mathsf{u}}$); that is, $!_X$ is unique for any X.

In addition to (5.5), we assume the following equivalences between terms:

$$\pi\langle p, p'\rangle = p \qquad \pi'\langle p, p'\rangle = p' \qquad \langle \pi r, \pi' r\rangle = r \qquad !_A\, t = \, !_B$$

These can be used to show that $\langle p, p'\rangle r = \langle pr, p'r\rangle$; this allows embedding a term within a context. For example,

$$f\langle g\langle \mathsf{left}, \mathsf{right}\rangle, \mathsf{accum}\rangle = f\langle \mathsf{left}, h\, f\langle \mathsf{right}, \mathsf{accum}\rangle\rangle$$

generalizes

$$\mathsf{flatten}\langle \mathsf{Tree}\langle \mathsf{left}, \mathsf{right}\rangle, \mathsf{accum}\rangle = \mathsf{flatten}\langle \mathsf{left}, \mathsf{flatten}\langle \mathsf{right}, \mathsf{accum}\rangle\rangle$$

and

$$\mathsf{squash}\langle \mathsf{Node}\langle \mathsf{left}, \langle \mathsf{info}, \mathsf{right}\rangle\rangle, \mathsf{accum}\rangle = \mathsf{squash}\langle \mathsf{left}, .\langle \mathsf{info}, \mathsf{squash}\langle \mathsf{right}, \mathsf{accum}\rangle\rangle\rangle$$

by the substitutions

$$\{f \mapsto \mathsf{flatten}, g \mapsto \mathsf{Tree}, h \mapsto \mathbf{1}\}$$

and

$$\{f \mapsto \mathsf{squash}, g \mapsto \mathsf{Node}\langle \pi, \langle \mathsf{info}!, \pi'\rangle\rangle, h \mapsto .\langle \mathsf{info}!, \mathbf{1}\rangle\}$$

We next consider matching cartesian combinator terms.

[10] Note that these form the arrows of a cartesian free category with finite products.

5.4.1 Matching

As shown by Bellegarde [Bel90], there may be an infinite number of matching substitutions between two cartesian combinator terms. For example, fg and AB (where $\mathsf{A}, \mathsf{B} \in \mathcal{C}_{X \to X}$) has three obvious matching substitutions, $[\mathsf{A}, \mathsf{B}]$, $[\mathbf{1}, \mathsf{AB}]$, and $[\mathsf{AB}, \mathbf{1}]$, but also substitutions such as $[\pi, \langle \mathsf{AB}, - \rangle]$ (where $-$ is a placeholder for any arbitrary term), $[\pi\pi, \langle\langle \mathsf{AB}, - \rangle, - \rangle]$, *etc.*[11] This is caused by allowing variables to be instantiated to terms which return products. Bellegarde solves this problem by characterizing solution sets using special variables. Another solution is to restrict $\mathcal{C}_{A \to B}$ and $\mathcal{V}_{A \to B}$ so that B is not a type of the form $X \times X'$. In our case, this is not a severe restriction since the language used in Focus does not allow functions to return (anonymous) products.[12] Besides making the problem finitary, it also means that the cartesian combinators are equivalent to second-order λ-terms. The result is that matching cartesian combinators is very similar to matching second-order λ-terms [HL78]. Some examples:

$$\begin{aligned}
[\mathsf{F}\langle \pi, \mathsf{Q}\pi' \rangle] &: \; g\langle \mathsf{P}, \mathbf{1} \rangle \;\to\; \mathsf{F}\langle \mathsf{P}, \mathsf{Q} \rangle \\
[\mathsf{F}\langle \mathsf{P}\pi', \mathsf{Q}\pi' \rangle] &: \; g\langle \mathsf{P}, \mathbf{1} \rangle \;\to\; \mathsf{F}\langle \mathsf{P}, \mathsf{Q} \rangle \\
[\mathsf{F}\langle \mathbf{1}, \mathbf{1} \rangle] &: \; f\mathsf{P} \;\to\; \mathsf{F}\langle \mathsf{P}, \mathsf{P} \rangle
\end{aligned}$$

The first example shows how π, π', and $\langle , \rangle$ are used to embed a term into a context. The second shows that subterms can be ignored. The third illustrates copying a subterm. The following terms do *not* match:

$$g\mathsf{P} \;\not\to\; \mathsf{F}\langle \mathsf{P}, \mathsf{Q} \rangle$$

This is because there is no substitution from P to Q.

But while matching cartesian combinators is well-defined, the next section shows that generalizing them is not.

5.4.2 Generalization

Adding pairs is necessary for generalizing terms with non-monadic function symbols. But cartesian products are too unconstrained because using them means that generalization is not well-defined. This section explains why and identifies restrictions to make generalization well-defined.

We can show that

Theorem 5.34 Given $a, b \in \mathcal{T}_{X \to Y}$, there is no subset of $\mathbf{G}(a, b)$ satisfying Definition 5.9.

[11] Note that we are assuming that the types of f and g are being instantiated during matching.

[12] Fixing the types of f and g also gives a finite set of solutions, but is redundant with disallowing functions that return products. Lemma B.11 shows that disallowing functions which return products is necessary to make generalization well-defined.

This result is proven in Appendix A, but the basis of the proof is illustrated by considering generalizations reachable from $\langle [\pi] : f\langle a, b\rangle \to a, [\pi'] : f\langle a, b\rangle \to b\rangle$. Let c be a term in $\mathcal{T}_{\mathsf{u}\to X}$; then the commutative diagram

$$\begin{tikzcd}
 & f\langle a, b\rangle \arrow[dl, "{[\pi]}"'] \arrow[d, "{[f\langle f\langle \pi, c!\rangle, f\langle c!, \pi'\rangle\rangle]}"] \arrow[dr, "{[\pi']}"] & \\
a & f\langle f\langle a, c!\rangle, f\langle c!, b\rangle\rangle \arrow[l, "{[\pi]}"'] \arrow[r, "{[\pi']}"] & b
\end{tikzcd}$$

has no upward generalization morphism. Further, a morphism of the form

$$\{f \mapsto f\langle f\langle \pi, c!\rangle, f\langle c!, \pi'\rangle\rangle\}$$

can be applied to *every* generalization reachable from

$$\langle [\pi] : f\langle a, b\rangle \to a, [\pi'] : f\langle a, b\rangle \to b\rangle$$

This means that there is an infinite supply of connected generalizations in $\mathrm{MSG}(a, b)$. Since the minimality condition of Definition 5.9 implies that the only generalization morphisms in $\mathrm{MSG}(a, b)$ must be identity substitutions, generalizing cartesian combinator terms—and second-order λ-terms—is not well-defined.

The problem is that cartesian combinators allow subterms to be ignored. For example, $[\pi] : f\langle \mathsf{a}, \mathsf{b}\rangle \to \mathsf{a}$ ignores the second argument to f. Likewise, $[\mathsf{b}!] : h\,\mathsf{a}! \to \mathsf{b}!$ ignores the argument a. Intuitively, a generalization should contain only those subterms that appear in both instances. That is, given $\langle \theta_1 : t \to a, \theta_2 : t \to b\rangle$, both θ_1 and θ_2 should use all of t in order to construct both a and b. This problem leads us to considering the *relevant combinators* (*cf.* [Mac71, Jac92]). These are combinators which necessarily preserve all subterms during application. We show that using relevant combinators gives well-defined, computable, maximally specific generalizations. The significance of choosing the relevant combinators is that relevant combinator terms are a proper subset of cartesian combinator terms in the sense that for every relevant combinator term there is a corresponding cartesian combinator terms but not vice versa. The next section introduces relevant combinator terms and shows that they lead to well-defined and useful generalizations.

5.5 Generalization with Relevant Combinator Terms

Using relevant combinators represents an alternative approach to introducing pairs of terms. In place of the combinators $\langle,\rangle$, π, π', and $!_X$, we introduce combinators which rearrange terms without deleting subterms. These combinators are known as the *relevant combinators* because of their relationship to *relevant logic* (*cf.* [Rea88, Jac92]), a logic based on the premise that an argument is valid only if all the assumptions made are relevant to the conclusion. This section describes the relevant combinators and discusses matching and generalizing relevant combinator terms. We then revise the definition of MSC to account for pairs and extend Algorithm 5.27 to compute such generalizations. Finally, Section 5.6 shows that these generalizations are useful in replay.

For relevant combinators, we denote pair types as $A \cdot B$ rather than $A \times B$ to be neutral about what kind of products are involved. To support rearrangement, we expect a number of types constructed from $\cdot$ and u to be isomorphic. For example,

$$\begin{array}{rcl} A \cdot (B \cdot C) & \cong & (A \cdot B) \cdot C \\ A \cdot \mathsf{u} & \cong & A \\ \mathsf{u} \cdot A & \cong & A \\ A \cdot B & \cong & B \cdot A \\ (A \cdot B) \cdot (C \cdot D) & \cong & (A \cdot C) \cdot (B \cdot D) \end{array}$$

All of these isomorphisms exist in the cartesian product structure. For example,

$$\langle \langle \pi, \pi\pi' \rangle, \pi'\pi' \rangle : A \times (B \times C) \to (A \times B) \times C$$

and

$$\langle \pi\pi, \langle \pi'\pi, \pi'\pi' \rangle \rangle : (A \times B) \times C \to A \times (B \times C)$$

are inverses. However, the cartesian combinator terms achieve them by repeatedly duplicating and discarding information. But the isomorphisms do not really involve duplication and discarding, merely rearrangement.

To express such rearrangements, we propose a notation called *restructors*.[13] A restructor is a pair of two *patterns*, written as $p \mapsto q$, where each pattern may contain variable symbols $x, y, \ldots$, a special symbol $\mathbf{1}_{\mathsf{u}}$, and the binary operator "$\cdot$". The following examples illustrate restructors:

$$\begin{array}{rclcl} x \cdot (y \cdot z) & \mapsto & (x \cdot y) \cdot z & : & A \cdot (B \cdot C) \to (A \cdot B) \cdot C \\ x \cdot \mathbf{1}_{\mathsf{u}} & \mapsto & x & : & A \cdot \mathsf{u} \to A \\ (x \cdot y) \cdot (z \cdot w) & \mapsto & (x \cdot z) \cdot (y \cdot w) & : & (A \cdot B) \cdot (C \cdot D) \to (A \cdot C) \cdot (B \cdot D) \end{array}$$

[13]These are closely related to KM-*graphs* [KM71, BCST93].

Note that restructors are functions and there is an obvious notion of type-correctness for them. Further, every variable occurring in a restructor must occur precisely once to the left of $\mapsto$. By considering different classes of restructors, we obtain different kinds of product structures:

1. *Linear combinator terms* or *symmetric monoidal combinator terms*: Every variable occurring in a restructor occurs *precisely once* to the right of $\mapsto$. One assumes that all restructors between two given types A and B are equal.

2. *Affine combinator terms*: Every variable occurring in a restructor occurs *at most once* to the right of $\mapsto$. Note that we can define the projection and ! operators as

$$\pi = (x \cdot y) \mapsto x,\ \pi' = (x \cdot y) \mapsto y,\ \text{and } ! = x \mapsto \mathsf{1}_{\mathsf{u}}$$

3. *Relevant combinator terms*: Every variable occurring in a restructor occurs *at least once* to the right of $\mapsto$. Note that we can define a pairing combinator based on the restructor $x \mapsto x \cdot x$ which duplicates terms:

$$\langle r, s\rangle = (r \cdot s)(x \mapsto x \cdot x)$$

4. *Cartesian combinator terms*: no restrictions on variables.

Thus the four classes of combinator terms are related as follows [Jac92]:

affine	=	symmetric monoidal + projections
relevant	=	symmetric monoidal + duplication
cartesian	=	symmetric monoidal + projections + duplication

More formally, each side of a restructor is a pattern in the least set of terms consistent with the relation $:_P$ defined as

$$\frac{s :_P X \qquad t :_P Y}{s \cdot t :_P X \cdot Y} \qquad\qquad \frac{x \in \mathcal{V}_X}{x :_P X} \qquad\qquad \frac{}{\mathsf{1}_{\mathsf{u}} :_P \mathsf{u}}$$

We say that a restructor is *relevant* if each variable in the left-hand side occurs in the right-hand side.

Given $:_P$, we define the relevant combinator terms $\mathcal{T}_{A \to B}$ as the least set of terms consistent with (5.4) and the rules

$$\frac{p :_P X \quad q :_P Y \quad \mathcal{FV}(p) = \mathcal{FV}(q) \quad \mathit{linear}(p)}{p \mapsto q : X \to Y} \qquad \frac{s : X_1 \to Y_1 \quad t : X_2 \to Y_2}{s \cdot t : (X_1 \cdot X_2) \to (Y_1 \cdot Y_2)}$$

where *linear*(p) is true if each variable in p occurs only once. To reduce the number of parentheses, we assume that function composition binds more closely than $\cdot$ and $\cdot$ more closely than $\mapsto$. To maintain the distinction between u and other types, and to avoid complications with defining composition, we assume $\mathcal{C}_{X \to \mathsf{u}}$ and $\mathcal{V}_{X \to \mathsf{u}}$ are empty.

In addition to (5.5), we assume the following equivalences between terms:

$$\begin{array}{rcll}
(s_1 \cdot s_2)(t_1 \cdot t_2) & = & s_1 t_1 \cdot s_2 t_2 & \\
\mathbf{1}_X & = & p \mapsto p & \text{where } p :_P X \\
p \mapsto q & = & \sigma(p) \mapsto \sigma(q) & \text{if } \mathit{linear}(\sigma(p)) \\
(p \mapsto q)t & = & \theta(q)(r \mapsto \sigma(q)) & \text{if } \theta(p) = t \text{ and } \sigma(p) = r \\
(p \mapsto q)(r \mapsto p) & = & r \mapsto q & \\
(p_1 \mapsto q_1) \cdot (p_2 \mapsto q_2) & = & p_1 \cdot p_2 \mapsto q_1 \cdot q_2 & \text{if } \mathit{linear}(p_1 \cdot p_2)
\end{array} \tag{5.6}$$

In the above, σ is a mapping from pattern variables to patterns and θ is a mapping from pattern variables to terms. Note that the first and second rules imply $\mathbf{1}_X \cdot \mathbf{1}_Y = \mathbf{1}_{X \cdot Y}$.

Examples 5.35 Let $\mathsf{A}, \mathsf{B} \in \mathcal{C}_{X \to Y}$ and $\mathsf{C} \in \mathcal{C}_{Y \to Z}$. Then

$$\begin{array}{rcl}
(z \mapsto \mathbf{1}_\mathsf{u} \cdot z)\mathsf{CA} & = & (\mathbf{1}_\mathsf{u} \cdot \mathsf{CA})(x \mapsto \mathbf{1}_\mathsf{u} \cdot x) \\
(x \mapsto x \cdot x)(x \cdot \mathbf{1}_\mathsf{u} \mapsto x) & = & x \cdot \mathbf{1}_\mathsf{u} \mapsto x \cdot x \\
(\mathbf{1}_\mathsf{u} \cdot (y \cdot y') \mapsto (y \cdot y') \cdot y)(\mathbf{1}_\mathsf{u} \cdot (\mathsf{A} \cdot \mathsf{B})) & = & ((\mathsf{A} \cdot \mathsf{B}) \cdot \mathsf{A})(\mathbf{1}_\mathsf{u} \cdot (x \cdot x') \mapsto (x \cdot x') \cdot x) \\
(y \cdot y' \mapsto y' \cdot y)(\mathsf{A} \cdot (\mathsf{B} \cdot \mathsf{B}))(x \mapsto x \cdot (x \cdot x)) & = & ((\mathsf{B} \cdot \mathsf{B}) \cdot \mathsf{A})(x \cdot x' \mapsto x' \cdot x)(x \mapsto x \cdot (x \cdot x)) \\
& = & ((\mathsf{B} \cdot \mathsf{B}) \cdot \mathsf{A})(x \mapsto (x \cdot x) \cdot x)
\end{array}$$

Relevant restructors cannot delete subterms (such as changing $\mathsf{A} \cdot \mathsf{B}$ to B) because of the restrictions that p is linear and each free variable in p must occur in q. Thus $x \cdot y \mapsto x$ (that is, the cartesian combinator π) is not a relevant restructor because it deletes the subterm matched by y. Deleting occurrences of $\mathbf{1}_\mathsf{u}$ is allowed, but this is primarily for consistency with category theory. Since by assumption the only function with the output type u is $\mathbf{1}_\mathsf{u}$, deleting occurrences of $\mathbf{1}_\mathsf{u}$ is not used often in generalization. In practice, the more typical operation is to insert occurrences of $\mathbf{1}_\mathsf{u}$. Since we use u to represent closed terms as with cartesian combinators, inserting occurrences of $\mathbf{1}_\mathsf{u}$ is used to embed a closed term into another term. For example,

$$\mathsf{F}(\mathbf{1} \cdot \mathsf{b})(x \mapsto x \cdot \mathbf{1}_\mathsf{u})\mathsf{a} \quad = \quad \mathsf{F}(\mathsf{a} \cdot \mathsf{b})(\mathbf{1}_\mathsf{u} \mapsto \mathbf{1}_\mathsf{u} \cdot \mathbf{1}_\mathsf{u}) \tag{5.7}$$

embeds a into the context $\mathsf{F}(- \cdot \mathsf{b})$.

Writing terms such as $(x \mapsto x \cdot \mathbf{1}_{\mathsf{u}})$ is tedious, and so we often use the following abbreviations (assuming $p :_P X$, $q :_P Y$, and $r :_P Z$):

$$\begin{array}{llcrcl}
\text{associate left:} & \alpha_{X,Y,Z} & = & p \cdot (q \cdot r) & \mapsto & (p \cdot q) \cdot r \\
\text{associate right:} & \alpha^{-1}_{X,Y,Z} & = & (p \cdot q) \cdot r & \mapsto & p \cdot (q \cdot r) \\
\text{delete left:} & \lambda_X & = & \mathbf{1}_{\mathsf{u}} \cdot p & \mapsto & p \\
\text{insert left:} & \lambda^{-1}_X & = & p & \mapsto & \mathbf{1}_{\mathsf{u}} \cdot p \\
\text{delete right:} & \varrho_X & = & p \cdot \mathbf{1}_{\mathsf{u}} & \mapsto & p \\
\text{insert right:} & \varrho^{-1}_X & = & p & \mapsto & p \cdot \mathbf{1}_{\mathsf{u}} \\
\text{commute:} & \gamma_{X,Y} & = & p \cdot q & \mapsto & q \cdot p \\
\text{duplicate:} & \delta_X & = & p & \mapsto & p \cdot p
\end{array}$$

We usually drop the type subscripts when they can be inferred from the context. Thus Equation 5.7 can be written as

$$\mathsf{F}(\mathbf{1} \cdot \mathsf{b})\varrho^{-1}\mathsf{a} \;\; = \;\; f(\mathsf{a} \cdot \mathsf{b})\delta$$

These abbreviations reflect the standard combinators for symmetric monodial categories [Mac71, Jac92]. For convenience, we give the set of all such restructors a name:

Definition 5.36 Define the set $\mathcal{R}$ be the terms generated by the production

$$R \; ::= \; \mathbf{1} \mid \alpha \mid \alpha^{-1} \mid \lambda \mid \lambda^{-1} \mid \varrho \mid \varrho^{-1} \mid \gamma \mid \delta \mid RR \mid R \cdot R$$

Clearly any $t \in \mathcal{R}$ can be represented by a unique $p \mapsto q$ modulo the sets of equivalences (5.5) and (5.6). Since the equations

$$\begin{array}{rr}
(\alpha \cdot \mathbf{1})\alpha(\mathbf{1} \cdot \alpha) = \alpha\alpha & (\varrho \cdot \mathbf{1})\alpha = \mathbf{1} \cdot \lambda \\
\lambda_{\mathsf{u}} = \varrho_{\mathsf{u}} & \gamma\gamma = \mathbf{1} \\
\lambda\gamma = \varrho & (\gamma \cdot \mathbf{1})\alpha(\mathbf{1} \cdot \gamma) = \alpha\gamma\alpha \\
\alpha(\mathbf{1} \cdot \delta)\delta = (\delta \cdot \mathbf{1})\delta & \gamma\delta = \delta \\
\lambda_{\mathsf{u}}\delta_{\mathsf{u}} = \mathbf{1}_{\mathsf{u}} & \varrho_{\mathsf{u}}\delta_{\mathsf{u}} = \mathbf{1}_{\mathsf{u}}
\end{array}$$

are satisfied, [Mac71] gives us the opposite direction:

Theorem 5.37 (Coherence) For every relevant $p \mapsto q$, there is an $r \in \mathcal{R}$ such that $r = p \mapsto q$.

One consequence of Theorem 5.37 is that the relevant combinator terms are a subset of the cartesian combinator terms. Let

$$\begin{array}{ccc} s \cdot t = \langle s\pi, t\pi' \rangle & \alpha = (\pi \cdot \pi\pi') \cdot \pi'\pi' & \lambda_X = \pi_{\mathsf{u},X} \\ \gamma = \langle \pi', \pi \rangle & \delta = \langle \mathbf{1}, \mathbf{1} \rangle & \varrho_X = \pi_{X,\mathsf{u}} \end{array} \tag{5.8}$$

Because these satisfy the preconditions for Theorem 5.37 as well, we can show that there is a cartesian combinator term corresponding to every relevant combinator term. The fact that $x \cdot y \mapsto x$ is not relevant means that the relevant combinator terms are a *proper* subset of the cartesian combinator terms.

One of the advantages of relevant combinator terms is that they can be drawn as graphs with composition denoted by a lines and pairs by diamonds, as shown in Figure 5.3. Restructors can be drawn as boxes with labeled inputs and outputs, as shown in Figure 5.4. The graph notation also leads to the algorithm for matching relevant combinator terms. This is explained in the next section.

5.5.1 Matching

Because of the ϱ and λ combinators and their inverses, matching relevant terms has the same difficulty as cartesian terms: there may be an infinite number of matches. Consider matching fg to AB where $\mathsf{A}, \mathsf{B} : \mathcal{C}_{X \to X}$. The three obvious substitutions are identical to those for monadic combinator terms: $[\mathsf{A}, \mathsf{B}]$, $[\mathbf{1}, \mathsf{AB}]$, $[\mathsf{AB}, \mathbf{1}]$. But because we have pairs, we also have an infinite number of other matching substitutions: $[\varrho, \mathsf{AB} \cdot \mathbf{1}_\mathsf{u}]$, $[\varrho\varrho, (\mathsf{AB} \cdot \mathbf{1}_\mathsf{u}) \cdot \mathbf{1}_\mathsf{u}]$, $[\varrho\lambda, \mathbf{1}_\mathsf{u} \cdot (\mathsf{AB} \cdot \mathbf{1}_\mathsf{u})]$, *etc.* As with cartesian combinator terms, we make matching finitary by assuming that the output types of constants and variables are not pairs.

Since there is a cartesian combinator term for every relevant combinator term, matching relevant combinator terms is decidable and finitary. By applying (5.8), we can reduce matching relevant combinator terms to matching cartesian combinator terms. If each term in a resulting substitution corresponds to a relevant combinator term, the terms can be obtained by applying the equalities

$$\langle s, t \rangle = (s \cdot t)\delta \qquad \pi = x \cdot y \mapsto x \qquad \pi' = x \cdot y \mapsto y \qquad ! = x \mapsto \mathbf{1}_\mathsf{u}$$

and then rules (5.5) and (5.6). An alternative algorithm which does not depend on using cartesian combinators is given in Appendix B.

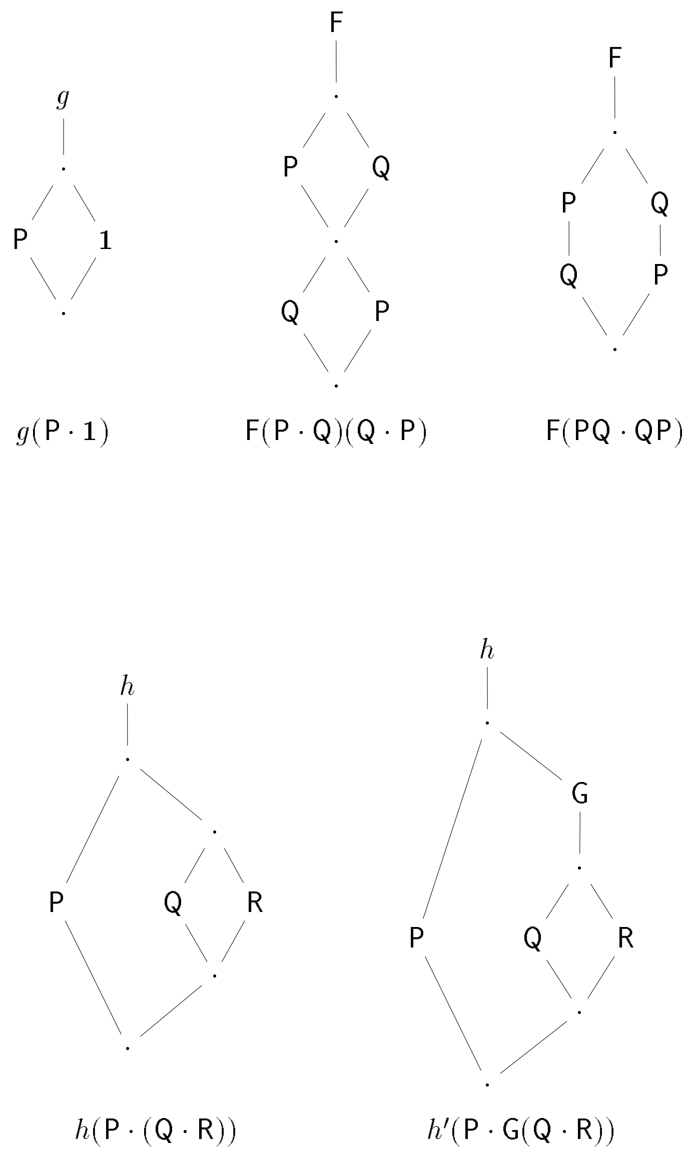
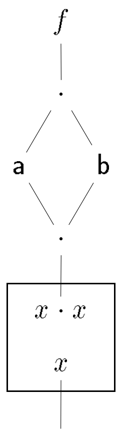
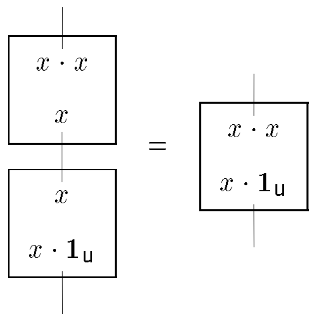

$g(\mathsf{P} \cdot \mathbf{1})$
$\mathsf{F}(\mathsf{P} \cdot \mathsf{Q})(\mathsf{Q} \cdot \mathsf{P})$
$\mathsf{F}(\mathsf{PQ} \cdot \mathsf{QP})$
$h(\mathsf{P} \cdot (\mathsf{Q} \cdot \mathsf{R}))$
$h'(\mathsf{P} \cdot \mathsf{G}(\mathsf{Q} \cdot \mathsf{R}))$

Figure 5.3: Examples of relevant combinator terms drawn as graphs.

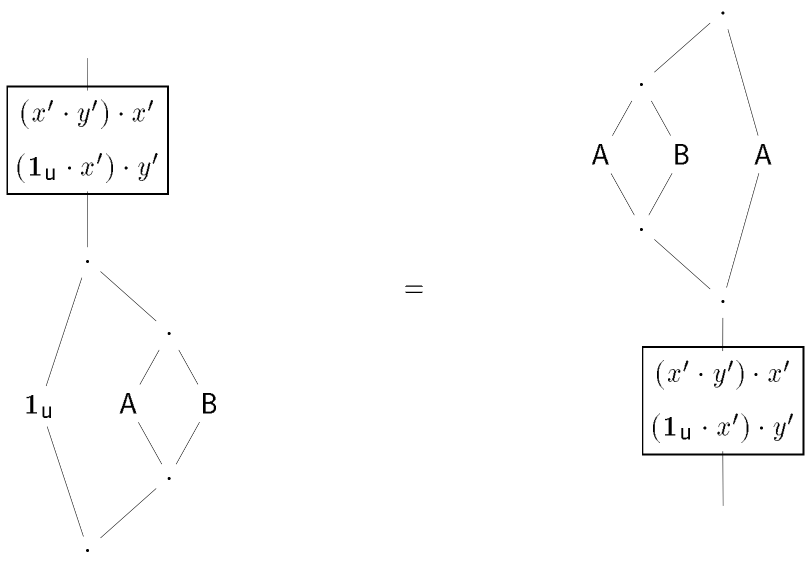


$f(\mathsf{a} \cdot \mathsf{b})\delta$



$x \cdot x$
x
x
$x \cdot \mathbf{1}_{\mathsf{u}}$
$=$
$x \cdot x$
$x \cdot \mathbf{1}_{\mathsf{u}}$

$(x \mapsto x \cdot x)(x \cdot \mathbf{1}_{\mathsf{u}} \mapsto x) = x \cdot \mathbf{1}_{\mathsf{u}} \mapsto x \cdot x$

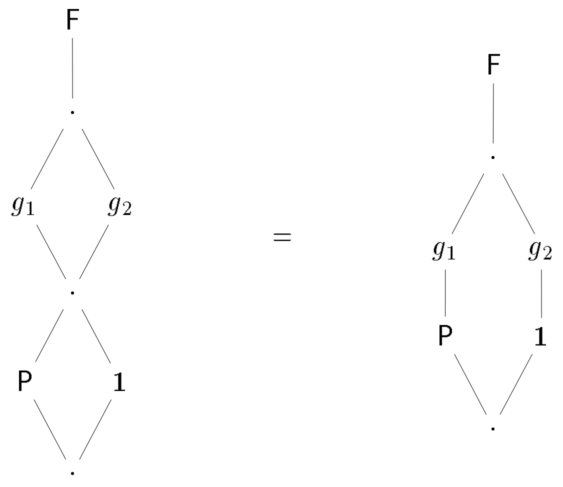


$(x' \cdot y') \cdot x'$
$(\mathbf{1}_{\mathsf{u}} \cdot x') \cdot y'$
$\mathbf{1}_{\mathsf{u}}$
A
B
$=$
A
B
A
$(x' \cdot y') \cdot x'$
$(\mathbf{1}_{\mathsf{u}} \cdot x') \cdot y'$

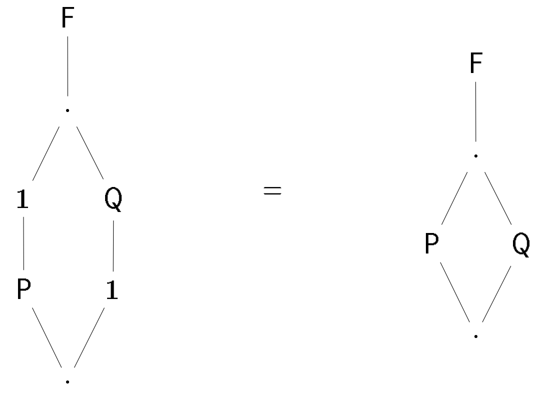
$$(\mathbf{1}_{\mathsf{u}} \cdot (x' \cdot y') \mapsto (x' \cdot y') \cdot x')(\mathbf{1}_{\mathsf{u}} \cdot (\mathsf{A} \cdot \mathsf{B})) \quad = \quad ((\mathsf{A} \cdot \mathsf{B}) \cdot \mathsf{A})(\mathbf{1}_{\mathsf{u}} \cdot (x \cdot y) \mapsto (x \cdot y) \cdot x)$$

Figure 5.4: Examples of relevant combinator terms with restructors drawn as boxes.

We can also view matching relevant combinator terms as filling in trees. For example, to match $f(\mathsf{P} \cdot \mathbf{1})$ to $\mathsf{F}(\mathsf{P} \cdot \mathsf{Q})$, we first replace f by $\mathsf{F}(g_1 \cdot g_2)$ to get



and then g_1 by $\mathbf{1}$ and g_2 by Q to get



Unfortunately, using graphs to represent terms is verbose, so we cannot do so in the generalization examples. But the reader is encouraged to apply this representation when verifying examples.

5.5.2 Generalization

Because relevant combinators cannot delete subterms, we can show that they give the same results as the monadic combinators. This section gives results for MSG, extends the definition of adjacency to include pairs, and gives an algorithm for computing MSC for ground terms.

Just as for monadic combinator terms, we can show:

Theorem 5.38 If a and b are relevant combinator terms having the same type, then $\mathrm{MSG}(a, b)$ exists.

The proof closely follows that for Theorem 5.16; it is given in Appendix B. Relevant combinators also follow monadic combinators with respect to adjacent variables. That is, adjacent variables result in many generalizations and ineffective transforms. However, consider the following extension to the definition of adjacency:

Definition 5.39 A pair of free variables f and g in term t are *adjacent* if there is a subterm fs in t and a relevant restructor $p \mapsto q$ such that $f(p \mapsto q)s = f(g \cdot s')$ for some s'.

That is, f and g are adjacent if they are not separated by some constant. For example, f and g are adjacent in fg, $f(s' \cdot g)$, and $f(r \cdot (g \cdot r'))$. With this definition of adjacency, Definitions 5.20 and 5.21 can again be used to identify a set of condensed generalizations which are computable and useful. Recall that Theorem 5.38 applies to $\mathbf{CG}(a, b)$ as well as $\mathbf{G}(a, b)$ since any morphism between generalizations in $\mathbf{CG}(a, b)$ is a morphism between generalizations in $\mathbf{G}(a, b)$.

All examples in 5.23 hold for relevant combinator terms as well. In addition, we have the following:

Example 5.40 $\mathrm{MSC}(\mathsf{F}(\mathsf{a}, \mathsf{G}(\mathsf{a}, \mathsf{b}), \mathsf{a}), \mathsf{F}(\mathsf{c}, \mathsf{H}(\mathsf{b}, \mathsf{c}), \mathsf{c}))$ is

$$\begin{tikzcd}
& \mathsf{F}(x \cdot f(\mathsf{b}) \cdot x) \arrow[dl, "{[\mathsf{a}, \mathsf{G}(\mathsf{a} \cdot \mathbf{1})\lambda^{-1}]}"'] \arrow[dr, "{[\mathsf{c}, \mathsf{H}(\mathbf{1} \cdot \mathsf{c})\varrho^{-1}]}"] & \\
\mathsf{F}(\mathsf{a} \cdot \mathsf{G}(\mathsf{a} \cdot \mathsf{b}) \cdot \mathsf{a}) & & \mathsf{F}(\mathsf{c} \cdot \mathsf{H}(\mathsf{b} \cdot \mathsf{c}) \cdot \mathsf{c})
\end{tikzcd}$$

To obtain generalization (5.1) proposed in Section 5.1, we would need to allow the more relaxed definition of relevance discussed in Section 5.3.3. This is discussed further in the next chapter.

The following example shows that there are pairs of terms with an exponential number of condensed generalizations:
Example 5.41

$$\mathrm{MSC}(\mathsf{F}(\mathsf{HA}_1 \cdots (\mathsf{HA}_{n-1} \cdot \mathsf{HA}_n)),\ \mathsf{G}(\mathsf{HB}_1 \cdots (\mathsf{HB}_{n-1} \cdot \mathsf{HB}_n)))$$

contains $n!$ generalizations formed by varying which A_i is matched to which B_j.

More examples of condensed generalizations are given in Section 5.6.

Computing $\mathrm{MSC}(a, b)$ is the same as in Algorithm 5.27 except for changing the definition of renaming and for changing the **Factor** step to support pairs. In Section 5.1.3, we called substitution $x \mapsto t$ a renaming if t is a free variable y. For relevant combinator terms, this needs to be generalized:

Definition 5.42 The substitution $x \mapsto t$ is a *renaming* if t is the form $y\tau$ for some free variable y and some relevant restructor τ for which there is a relevant restructor τ^{-1} with $\tau\tau^{-1} = \mathbf{1}$ and $\tau^{-1}\tau = \mathbf{1}$.

For the **Factor** step, suppose a constant K appears in both bindings for some variable f. K might be embedded in some pair of the form $s_1(\cdots K \cdots)s_2$. To separate K from the context, we use the equivalence relations for relevant combinator terms to obtain a term of the form $s_1'(K \cdot \mathbf{1})s_2'$ where $\mathbf{1}$ "passes through" any data produced by s_2' and used by s_1' but not K. For instance, $\mathsf{A}(\mathsf{B}(K \cdot \mathsf{C}) \cdot \mathsf{D})\mathsf{E}$ is equivalent to $\mathsf{A}(\mathsf{B}(\mathbf{1} \cdot \mathsf{C}) \cdot \mathsf{D})\alpha(K \cdot \mathbf{1})\alpha^{-1}\mathsf{E}$ as shown in Figure 5.5. This gives us the rule

$$\frac{t,\ \theta_1 \cup \{f \mapsto r(K \cdot \mathbf{1})r'\},\ \theta_2 \cup \{f \mapsto s(K \cdot \mathbf{1})s'\}}{\{f \mapsto h(K \cdot \mathbf{1})h'\}(t),\ \theta_1 \cup \{h \mapsto r, h' \mapsto r'\},\ \theta_2 \cup \{h \mapsto s, h' \mapsto s'\}}$$

Theorem 5.43 Given $t \in \mathcal{T}_{X \to Y}$ and a constant K in t, there are r and s such that $t = r(K \cdot \mathbf{1})s$.

Proof Let t' be t with $\underline{K}$ (a fresh symbol) substituted for the occurrence of K we wish to isolate. Then repeatedly apply the following rewrite rules (without applying any of the equivalences for relevant combinator terms) until no rule applies:

$$\begin{array}{rcll}
r\,\underline{K}\,s & \to & r\varrho(\underline{K} \cdot \mathbf{1}_{\mathsf{u}})\varrho^{-1}s & \\
r(\underline{K} \cdot t)s & \to & r(\mathbf{1} \cdot t)(\underline{K} \cdot \mathbf{1})s & \text{if } t \neq \mathbf{1} \\
r(t_1 \cdot t_2)s & \to & r\gamma(t_2 \cdot t_1)\gamma s & \text{if } \underline{K} \text{ occurs in } t_2 \\
r(t_1t_2 \cdot t_3)s & \to & r(t_1 \cdot \mathbf{1})(t_2 \cdot t_3)s & \text{if } \underline{K} \text{ occurs in } t_1 \text{ or } t_2 \\
 & & & \quad \text{where } t_1 \text{ and } t_2 \neq \mathbf{1} \\
r((t_1 \cdot t_2) \cdot t_3)s & \to & r\alpha(t_1 \cdot (t_2 \cdot t_3))\alpha^{-1}s & \text{if } \underline{K} \text{ occurs in } t_1 \\
r((t_1 \cdot t_2) \cdot t_3)s & \to & r(\gamma \cdot \mathbf{1})((t_2 \cdot t_1) \cdot t_3)(\gamma \cdot \mathbf{1})s & \text{if } \underline{K} \text{ occurs in } t_2
\end{array}$$

We can use the position of $\underline{K}$ in t' to show that these rules terminate. □

However, this version of **Factor** is not complete. K may appear more than once in $\theta_1(f)$ or $\theta_2(f)$, and the step must allow all occurrences to be factored at the same time. This is necessary to support generalizations such as

$$\mathsf{A}(\mathsf{K} \cdot \mathsf{K}) \xleftarrow{[\mathsf{A}\delta]} f\mathsf{K} \xrightarrow{[\mathsf{B}]} \mathsf{BK} \tag{5.9}$$

In general, if $\delta^n = x \mapsto (x \cdot (x \cdots (x \cdot x)))$ where there are n occurrences of x in the right-hand side (note that $\delta^1 = \mathbf{1}$), the **Factor** step must be refined to generalize terms of the form $r(\delta^n K \cdot \mathbf{1})s$.

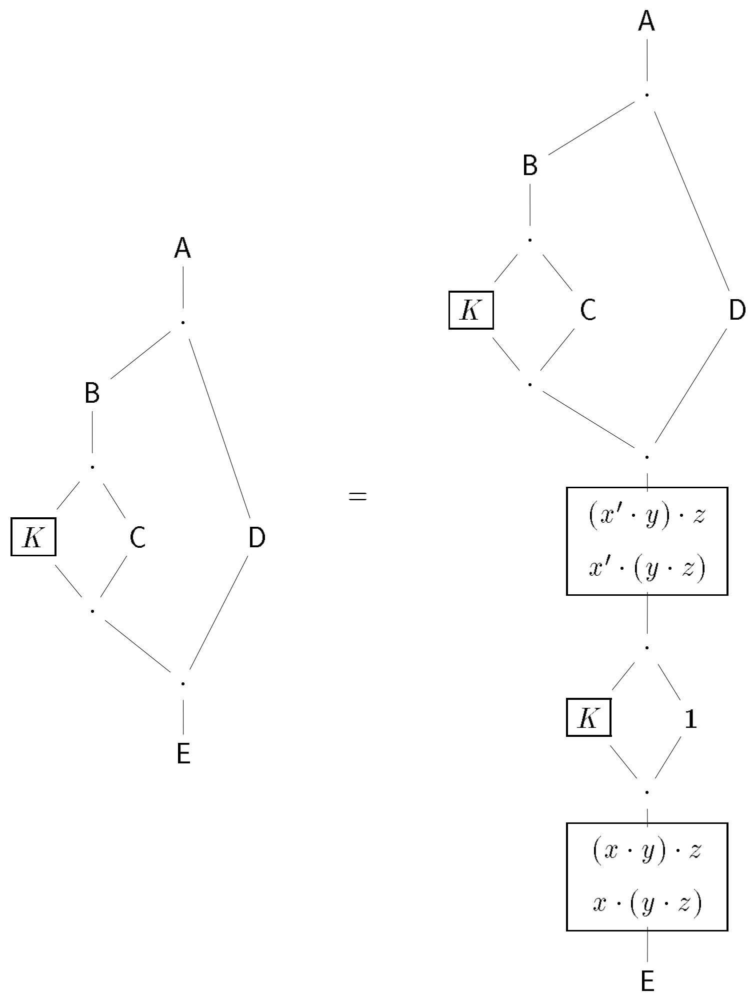


Figure 5.5: Isolating K from $\mathsf{A}(\mathsf{B}(K \cdot \mathsf{C}) \cdot \mathsf{D})\mathsf{E}$.

A second issue is that the occurrence of h' in the above version of **Factor** is not valid. The types of variables (and functions) are restricted to be of the type $X \to Y$ where Y is not a pair type. The variable h' in the above step violates this condition. But if s in $r(K \cdot \mathbf{1})s$ is not a restructor, it must be of the form $s_1 \cdot s_2$, so

$$r(K \cdot \mathbf{1})s = r(K \cdot \mathbf{1})(s_1 \cdot s_2) = r(\mathbf{1} \cdot s_2)(Ks_1 \cdot \mathbf{1})$$

Furthermore, if K has the type $X_1 \cdot (X_2 \cdots (X_{n-1} \cdot X_n) \ldots) \to Y$, then $s_1 = s_1 \cdot (s_2 \cdots (s_{n-1} \cdot s_n) \ldots)$. If K's type has some other parenthesization, we need to apply the appropriate combination of α and α^{-1} to reorganize the parenthesization. This is done by introducing restructors from the following language:

Definition 5.44 The *associative restructors*, $\mathcal{A}$, is the set of restructors generated by the production

$$A \ ::= \ \mathbf{1} \mid \alpha \mid \alpha^{-1} \mid AA \mid A \cdot A$$

If we let $\overline{s_n}$ represent the term $s_1 \cdot (s_1 \cdots (s_{n-1} \cdot s_n) \ldots)$, then

Theorem 5.45 Given $t \in \mathcal{T}_{X \to Y}$ and a subterm Ks with m occurrences in t, there are r, $\overline{s_n}$, $\mu \in \mathcal{A}$, and $\tau \in \mathcal{R}$ such that $t = r(\delta^m K\mu(\overline{s_n}) \cdot \mathbf{1})\tau$.

This follows by generalizing the proof of Theorem 5.43.

Algorithm 5.46 To compute MSC(a, b) where a and b are ground relevant combinator terms in $\mathcal{T}_{X \to Y}$, apply the steps of Algorithm 5.27 with **Merge** refined to use Definition 5.42 and **Factor** replaced by the following two rules:

Factor-Constant

$$\frac{t,\ \theta_1 \cup \{f \mapsto r(\delta^m K\mu(\overline{r_n}) \cdot \mathbf{1})\tau\},\ \theta_2 \cup \{f \mapsto s(\delta^{m'} K\mu(\overline{s_n}) \cdot \mathbf{1})\tau\}}{\begin{array}{l} \{f \mapsto h(K\mu(\overline{h_n}) \cdot \mathbf{1})\tau\}(t), \\ \theta_1 \cup \{h \mapsto r(\delta^m \ \cdot \mathbf{1}), h_1 \mapsto r_1, \ldots, h_n \mapsto r_n\}, \\ \theta_2 \cup \{h \mapsto s(\delta^{m'} \cdot \mathbf{1}), h_1 \mapsto s_1, \ldots, h_n \mapsto s_n\} \end{array}}$$

where $K \in \mathcal{C}_{X \to Y}$, $\mu \in \mathcal{A}$, $\tau \in \mathcal{R}$, and h and h_i are fresh variables. Note that τ must be the same in both $\theta_1(f)$ and $\theta_2(f)$; otherwise, $\theta_1(h)$ or $\theta_2(h)$ would not be a relevant combinator term.

Factor-Restructor Factoring restructors is similar to factoring constants except that we do not need to introduce the h_i's. By the assumption that no function returns a pair type, given

$$\begin{tikzcd}
 & f \arrow[d, "{[h(\mathsf{F}(h_1 \cdot h_2) \cdot \mathbf{1}_{\mathsf{u}})\varrho^{-1}]}"] \arrow[ddddl, "{[\mathsf{F}(\mathsf{G} \cdot \mathsf{G})]}"'] \arrow[ddddr, "{[\mathsf{F}(\mathsf{H} \cdot \mathsf{H})]}"] & \\
 & h(\mathsf{F}(h_1 \cdot h_2) \cdot \mathbf{1}_{\mathsf{u}})\varrho^{-1} \arrow[d, "{[\varrho, h_1, h_2]}"] \arrow[dddl, "{[\varrho, \mathsf{G}, \mathsf{G}]}"'] \arrow[dddr, "{[\varrho, \mathsf{H}, \mathsf{H}]}"] & \\
 & \mathsf{F}(h_1 \cdot h_2) \arrow[d, "{[h_1, h_1]}"] \arrow[ddl, "{[\mathsf{G}, \mathsf{G}]}"'] \arrow[ddr, "{[\mathsf{H}, \mathsf{H}]}"] & \\
 & \mathsf{F}(h_1 \cdot h_1) \arrow[dl, "{[\mathsf{G}]}"'] \arrow[dr, "{[\mathsf{H}]}"] & \\
\mathsf{F}(\mathsf{G} \cdot \mathsf{G}) & & \mathsf{F}(\mathsf{H} \cdot \mathsf{H})
\end{tikzcd}$$

Figure 5.6: Generalizing $\mathsf{F}(\mathsf{G}, \mathsf{G})$ and $\mathsf{F}(\mathsf{H}, \mathsf{H})$.

a $\tau \in \mathcal{R}$ in t, there is an r and a $\tau' \in \mathcal{R}$ such that $t = r(\tau \cdot \mathbf{1})\tau'$. But since τ' must be the same in both $\theta_1(h)$ and $\theta_2(h)$, we can write $(\tau \cdot \mathbf{1})\tau'$ as τ''. This gives us the rule

$$\frac{t,\ \theta_1 \cup \{f \mapsto r\tau\},\ \theta_2 \cup \{f \mapsto s\tau\}}{\{f \mapsto h\tau\}(t),\ \theta_1 \cup \{h \mapsto r\},\ \theta_2 \cup \{h \mapsto s\}}$$

where h is a fresh variable. For termination, we must add the condition that there be no $\tau^{-1} \in \mathcal{R}$ such that $\tau^{-1}\tau = \mathbf{1}$. That is, τ must contain δ_X in some form; τ cannot be (say) γ or δ_{u}.

Example 5.47 Figure 5.6 illustrates most of the steps of Algorithm 5.46. From top to bottom, the steps are **Factor-Constant, Delete,** and **Merge**. In the first step, $K = \mathsf{F}$, $\mu = \mathbf{1}$, $r = s = \varrho$, and $\tau = \varrho^{-1}$.

Example 5.48 Factoring A from $\mathsf{F}(\mathsf{A} \cdot (\mathsf{A} \cdot \mathsf{b}))(\mathbf{1} \cdot \varrho^{-1})$ and $\mathsf{G}(\mathsf{AC} \cdot \mathsf{AC})$, where $\mathsf{A}, \mathsf{C} \in \mathcal{C}_{X \to X}$ and $\mathsf{b} \in \mathcal{C}_{\mathsf{u} \to X}$, gives a more complex example of applying **Factor-Constant**. Since $\alpha(\mathbf{1} \cdot \varrho^{-1}) = \varrho^{-1}$,

$$\mathsf{F}(\mathsf{A} \cdot (\mathsf{A} \cdot \mathsf{b}))(\mathbf{1} \cdot \varrho^{-1}) = \mathsf{F}\alpha^{-1}(\mathbf{1} \cdot \mathsf{b})(\delta\mathsf{A} \cdot \mathbf{1}_{\mathsf{u}})\varrho^{-1}$$

and

$$\mathsf{G}(\mathsf{AC} \cdot \mathsf{AC}) = \mathsf{G}\varrho(\delta\mathsf{AC} \cdot \mathbf{1}_{\mathsf{u}})\varrho^{-1}$$

Applying **Factor-Constant** with $m = m' = 2$ gives

$$\frac{f, \{f \mapsto \mathsf{F}\alpha^{-1}(\mathbf{1}\cdot\mathsf{b})(\delta\mathsf{A}\cdot\mathbf{1})\varrho^{-1}\}, \{f \mapsto \mathsf{G}\varrho(\delta\mathsf{AC}\cdot\mathbf{1}_{\mathsf{u}})\varrho^{-1}\}}{h(\mathsf{A}h_1\cdot\mathbf{1}_{\mathsf{u}})\varrho^{-1}, \{h \mapsto \mathsf{F}\alpha^{-1}(\mathbf{1}\cdot\mathsf{b})(\delta\cdot\mathbf{1}), h_1 \mapsto \mathbf{1}\}, \{h \mapsto \mathsf{G}\varrho(\delta\cdot\mathbf{1}), h_1 \mapsto \mathsf{C}\}}$$

The cases where $m \neq m'$ violate the condition that τ be the same in both $\theta_1(f)$ and $\theta_2(f)$.

Example 5.49 We illustrate **Factor-Restructor** by generalizing

$$\mathsf{F}(x\cdot y \mapsto (x\cdot x)\cdot y) = \mathsf{F}(\delta\cdot\mathbf{1})$$

and

$$\mathsf{F}(x\cdot y \mapsto (y\cdot x)\cdot x) = \mathsf{F}\alpha\gamma(\delta\cdot\mathbf{1})$$

where $x, y :_P X$:

$$\frac{f,\ \{f \mapsto \mathsf{F}(\delta\cdot\mathbf{1})\},\ \{f \mapsto \mathsf{F}\alpha\gamma(\delta\cdot\mathbf{1})\}}{h(\delta\cdot\mathbf{1}),\ \{h \mapsto \mathsf{F}\},\ \{h \mapsto \mathsf{F}\alpha\gamma\}}$$

Note that **Factor-Constant** is *not* applicable at this point.

Thus generalization of relevant combinator terms is well-defined and computable. The next section shows that it is useful.

5.6 Applying Second-Order Generalization to Replay

The set $\mathrm{MSC}(a, b)$ identifies the possible syntactic matches between subterms of a and b. Given a particular generalization $\langle\theta_1 : t \to a, \theta_2 : t \to b\rangle$, we can build a set of transforms by identifying the components of θ_1 and θ_2:

$$\{r \Rightarrow s \mid f \mapsto r \in \theta_1 \text{ and } f \mapsto s \in \theta_2\}$$

These transforms can be used in two ways. The first use is to update subterms in derivations.[14] The second use is as a similarity metric: by comparing the size of two transform sets, we can decide which pair of terms represents the closest match by counting the number of constants in the transforms. Chapter 6 describes computing and using transforms in more detail. In the remainder of this chapter, we look at applying second-order generalization to replay problems described earlier in this thesis. We also show that using first-order generalization is inadequate.

The following two examples illustrate applying second-order generalization to the examples in Chapter 4. In each case, we assume that all instance terms are first-order and each function

[14] Note that we have not given a mechanism for applying transforms; more explicit algorithms are given in Chapter 6.

has the type $\iota \cdot \iota \cdots \iota \to \iota$ where ι is the generic type *item*; types are discussed further in Section 6.1.

Example 5.50 Consider Example 4.2 (p. 38), in which the proof of

$$\mathsf{len(fringe(tree))} == 0 \to \mathsf{False}$$

was used to show that

$$\mathsf{size(tree)} == 0 \to \mathsf{False}$$

MSC(len fringe tree, size tree) contains just

$$\begin{tikzcd}\mathsf{len\ fringe\ tree} & f\ \mathsf{tree} \arrow[l, "\mathsf{[len\ fringe]}"'] \arrow[r, "\mathsf{[size]}"] & \mathsf{size\ tree}\end{tikzcd}$$

which leads to the transform

$$\mathsf{len\ fringe} \Rightarrow \mathsf{size}$$

Applying this to the lemma

$$\mathsf{len(fringe(t))} == 0 \to \mathsf{False}$$

gives

$$\mathsf{size(t)} == 0 \to \mathsf{False}$$

as desired. This example is fairly simple, yet shows where rules obtained from first-order anti-unification may be overly specific. The first-order anti-unifier gives the transform

$$\mathsf{len(fringe(tree))} \Rightarrow \mathsf{size(tree)}$$

This transform does not match len(fringe(t)). While this problem could be solved with an appropriate heuristic for variables in terms, using the second-order transform is more general.

Example 5.51 Consider Example 4.1 (p. 36). Comparing the initial specification

$$\mathsf{flatten(tree \cdot accum) = append(fringe\ tree \cdot accum)}$$

to

$$\mathsf{squash(tree \cdot accum) = append(nodes\ tree \cdot accum)}$$

gives the transform fringe $\Rightarrow$ nodes. Applying this transform to the `expand` step updates it appropriately.

To match the cases, we generalize the possible matches. For the recursive case, the generalizations are

$$\begin{tikzcd}
\mathsf{flatten}(\mathsf{Tree}(\mathsf{l}\cdot\mathsf{r})\delta_{\mathsf{u}}\cdot\mathsf{a}) = \mathsf{append}(\mathsf{append}(\mathsf{fringe}\ \mathsf{l}\cdot\mathsf{fringe}\ \mathsf{r})\delta_{\mathsf{u}}\cdot\mathsf{a}) \\
f(\mathbf{1}_{\mathsf{u}}\cdot\mathsf{a}) = \mathsf{append}(w\cdot\mathsf{a}) \arrow[u, "{[\mathsf{flatten}(\mathsf{Tree}(\mathsf{l}\cdot\mathsf{r})\delta_{\mathsf{u}}\cdot\mathbf{1}_{\iota}),\ \mathsf{append}(\mathsf{fringe}\ \mathsf{l}\cdot\mathsf{fringe}\ \mathsf{r})\delta_{\mathsf{u}}]}"'] \arrow[d, "{[\mathsf{squash}(\mathsf{Tip}\cdot\mathbf{1}_{\iota}),\mathsf{Nil}]}"'] \\
\mathsf{squash}(\mathsf{Tip}\cdot\mathsf{a}) = \mathsf{append}(\mathsf{Nil}\cdot\mathsf{a})
\end{tikzcd}$$

and

$$\begin{tikzcd}
\mathsf{flatten}(\mathsf{Tree}(\mathsf{l}\cdot\mathsf{r})\delta_{\mathsf{u}}\cdot\mathsf{a}) = \mathsf{append}(\mathsf{append}(\mathsf{fringe}\ \mathsf{l}\cdot\mathsf{fringe}\ \mathsf{r})\delta_{\mathsf{u}}\cdot\mathsf{a}) \\
f((\mathsf{l}\cdot\mathsf{r})\cdot\mathsf{a}) = \mathsf{append}(\mathsf{append}(g\mathsf{l}\cdot h\mathsf{r})\delta_{\mathsf{u}}\cdot\mathsf{a}) \arrow[u, "{[\mathsf{flatten}(\mathsf{Tree}\cdot\mathbf{1}_{\iota}),\mathsf{fringe},\mathsf{fringe}]}"'] \arrow[d, "{[\mathsf{squash}(\mathsf{Node}(\mathbf{1}_{\iota}\cdot(\mathsf{i}\cdot\mathbf{1}_{\iota})\lambda^{-1})\cdot\mathbf{1}_{\iota}),\ \mathsf{nodes},.(\mathsf{i}\cdot\mathsf{nodes})\lambda^{-1}]}"'] \\
\mathsf{squash}(\mathsf{Node}(\mathsf{l}\cdot(\mathsf{i}\cdot\mathsf{r})\delta_{\mathsf{u}})\delta_{\mathsf{u}}\cdot\mathsf{a}) = \mathsf{append}(\mathsf{append}(\mathsf{nodes}\ \mathsf{l}\cdot.(\mathsf{i}\cdot\mathsf{nodes}\ \mathsf{r}))\delta_{\mathsf{u}}\cdot\mathsf{a})
\end{tikzcd}$$

By measuring the relative sizes of the resulting transforms, we obtain a metric for identifying the closest match. In the first generalization, the instance terms contain 19 function symbols and the transform 12. In the second generalization, the instance terms contain 27 function symbols while the transform just 11. Since $11/27 \approx 41\% < 12/19 \approx 63\%$, second-order generalization suggests that the second pair of terms are more closely related than the first pair, giving ReFocus the information it needs to match the recursive cases in the derivations. The base case gives similar results. In contrast, the first-order generalization is trivial: $x = y$. Since all of the function symbols appear in the resulting transforms, there is no basis for choosing one match over the other.

In general, we compare matches between terms by comparing the norms of their generalizations, where

Definition 5.52 Given generalization $g = \langle \theta_1 : t \to a, \theta_2 : t \to b\rangle \in \mathbf{G}(a, b)$, its *norm*, denoted $\|g\|$, is defined as

$$\|g\| = (|\theta_1| + |\theta_2|)/|t|$$

where $|t|$ is a measure of the size of a term and

$$|\theta| = \sum_{x \in dom\,\theta} |\theta(x)|$$

For this chapter, we define $|t|$ as the number of constants in t.

Finally, analogy is needed to pick the appropriate choice near the end of the script. For this example, assume $\cdot$ associates to the right, and let $@_k(f \cdot a_1 \cdots a_n)$ denote applying function f to arguments $a_1 \cdots a_n$. Also, assume that appropriate occurrences of δ_{u} are inserted in the term. We omit these to simplify the notation and since they can be inferred from the context.

The two generalizations are

$$\begin{tikzcd}
@_2(\mathsf{flatten} \cdot @_2(\mathsf{Tree} \cdot \mathsf{l} \cdot \mathsf{r}) \cdot \mathsf{a}) = @_2(\mathsf{flatten} \cdot \mathsf{l} \cdot @_2(\mathsf{flatten} \cdot \mathsf{r} \cdot \mathsf{a})) \\
@_2(f \cdot g(\mathsf{l} \cdot \mathsf{r}) \cdot \mathsf{a}) = @_2(f \cdot \mathsf{l} \cdot h\ @_2(f \cdot \mathsf{r} \cdot \mathsf{a})) \arrow[u, "{[\mathsf{flatten}, @_2(\mathsf{Tree} \cdot \mathbf{1}_\iota)\lambda^{-1}, \mathbf{1}_\iota]}"'] \arrow[d, "{\begin{array}{l}[\mathsf{squash}, @_3(\mathsf{Node} \cdot \mathbf{1}_\iota \cdot \mathsf{i} \cdot \mathbf{1}_\iota)(\mathbf{1}_\iota \cdot \mathbf{1}_\iota \cdot \lambda^{-1}), \\ \ @_2(. \cdot \mathsf{i} \cdot \mathbf{1}_\iota)(\mathsf{x} \mapsto \mathbf{1}_{\mathsf{u}} \cdot \mathbf{1}_{\mathsf{u}} \cdot \mathsf{x})]\end{array}}"'] \\
@_2(\mathsf{squash} \cdot @_3(\mathsf{Node} \cdot \mathsf{l} \cdot \mathsf{i} \cdot \mathsf{r}) \cdot \mathsf{a}) = @_2(\mathsf{squash} \cdot \mathsf{l} \cdot @_2(. \cdot \mathsf{i} \cdot @_2(\mathsf{squash} \cdot \mathsf{r} \cdot \mathsf{a})))
\end{tikzcd}$$

and

$$\begin{tikzcd}
@_2(\mathsf{flatten} \cdot @_2(\mathsf{Tree} \cdot \mathsf{l} \cdot \mathsf{r}) \cdot \mathsf{a}) = @_2(\mathsf{flatten} \cdot \mathsf{l} \cdot @_2(\mathsf{flatten} \cdot \mathsf{r} \cdot \mathsf{a})) \\
@_2(x \cdot f(\mathsf{l} \cdot \mathsf{r}) \cdot \mathsf{a}) = g(@_2(y \cdot h\mathsf{r} \cdot \mathsf{a})) \arrow[u, "{\begin{array}{l}[\mathsf{flatten}, @_2(\mathsf{Tree} \cdot \mathbf{1}_\iota)\lambda^{-1}, \\ \ @_2(\mathsf{flatten} \cdot \mathsf{l} \cdot \mathbf{1}_\iota)(x \cdot \mathbf{1}_{\mathsf{u}} \cdot \mathbf{1}_{\mathsf{u}} \cdot x), \mathsf{flatten}, \mathbf{1}_\iota]\end{array}}"'] \arrow[d, "{\begin{array}{l}[\mathsf{squash}, @_3(\mathsf{Node} \cdot \mathbf{1}_\iota \cdot \mathsf{i} \cdot \mathbf{1}_\iota)(\mathbf{1}_\iota \cdot \mathbf{1}_\iota \cdot \lambda^{-1}), \mathbf{1}_\iota, \\ \ \mathsf{append}, @_2(\mathsf{squash} \cdot \mathsf{l} \cdot @_2(. \cdot \mathsf{i} \cdot @_1(\mathsf{nodes} \cdot \mathbf{1}_\iota))) \\ \qquad (x \mapsto \mathbf{1}_{\mathsf{u}} \cdot \mathbf{1}_{\mathsf{u}} \cdot \mathbf{1}_{\mathsf{u}} \cdot \mathbf{1}_{\mathsf{u}} \cdot \mathbf{1}_{\mathsf{u}} \cdot x)]\end{array}}"'] \\
\begin{array}{l}@_2(\mathsf{squash} \cdot @_3(\mathsf{Node} \cdot \mathsf{l} \cdot \mathsf{i} \cdot \mathsf{r}) \cdot \mathsf{a}) = \\ \qquad @_2(\mathsf{append} \cdot @_2(\mathsf{squash} \cdot \mathsf{l} \cdot @_2(. \cdot \mathsf{i} \cdot @_1(\mathsf{nodes} \cdot \mathsf{r}))) \cdot \mathsf{a})\end{array}
\end{tikzcd}$$

Note that the $@_n$ notation allows the generalizations to capture common subterms without being redundant. Further justification for this notation is given in the next chapter. The norm of the first generalization is $10/32 \approx 31\%$, while the norm of the second is $20/34 \approx 59\%$. Thus second-order analogy suggests choosing the first choice since its structure is closer to the original result than the second, as desired.

5.7 Conclusion

Analogical reasoning is a vital part of replaying program derivations. In this chapter, we have given a definition of syntactic analogy based on generalization and showed that first-order generalization is not adequate for replay. This lead to using second-order generalization, which was defined by uniquely minimal complete set of generalizations.[15] We showed that such generalizations exist when we eliminate combinators which delete useful subterms. Further, we gave a semi-practical algorithm to compute such generalizations and showed that they are useful by applying them to replay problems from Chapters 3 and 4.

However, a number of issues remain:

1. How to integrate second-order generalization into a practical replay system. Algorithm 5.46 is expensive because it is highly non-deterministic, and so a more practical algorithm is needed.

2. How to construct transformation rules from generalizations. Both the representation of the rules and the mechanics of generating them need to be discussed.

3. Whether there a benefit to considering third-order or higher-order generalizations. Example 5.3 suggests that third-order generalizations may be useful since in this case the positions of A and and B are being swapped. But it is not clear how to define such generalizations or if such generalizations would be useful in practice.

While important, the last issue is not addressed in this thesis; it is left as future work. The next chapter addresses the first and second issues. We give a quadratic-time algorithm for computing generalizations (under certain assumptions) and discuss converting generalizations into useful sets of transforms.

[15] See Appendix C for an equivalent definition based on a categorical framework.

Chapter 6

The Practical Construction of Second-order Generalizations

Chapter 5 shows that analogy based on second-order generalization provides a flexibility that cannot be achieved by first-order generalization. Second-order variables abstract contexts, so analogies based on second-order generalizations can take advantage of similarities embedded within dissimilar contexts. However, while second-order generalization is flexible, it cannot be applied to replay in its pure form. There are several issues that must be addressed: restricting the number of generalizations, computing generalizations efficiently, generalizing terms from dissimilar domains, and ensuring that the resulting transformation rules are useful.

Restricting the number of generalizations is the most important issue. While the flexibility of second-order matching makes it necessary to allow multiple generalizations, replay can use only one at a time. Replay is too expensive to implement it to try all possible generalizations. Criteria are needed for selecting a best generalization when multiple ones are possible. Choosing a best generalization is also needed for efficiency. Computing all generalizations is expensive, while computing a subset allows optimizations which overlook some possibilities. Thus restricting the number of generalizations is necessary both to improve efficiency and to satisfy replay.

The second most important issue is computing generalizations more efficiently. Algorithm 5.46 is inefficient because the **Factor-Constant** step is highly nondeterministic. This leads to computing the same generalizations in multiple ways. By giving preference to certain types of generalizations, we change the problem so that dynamic programming (*cf.* [RND77]) can be used to reduce the amount of unnecessary computation. In effect, this gives us a variation on Algorithm 5.46 in which applications of the **Factor-Constant** step are chosen so as to favor particular generalizations. The result is an algorithm which still takes exponential time in the worst case, but is quadratic in the expected cases.

Whereas the above two issues concern efficiency, the remaining two issues concern usability. Generalizing terms from dissimilar domains, as done for derivation-by-analogy, means generalizing terms when the only thing they have in common is their basic structure. If the problem domains are very different, the derivations are likely to share very few symbols. This leads to transforms which are too specific in the same way that first-order generalizations are too specific. Yet some of the most useful applications of replay involve domains with different namespaces. Domains often share an underlying structure (in a theoretical sense), so a derivation may be more similar to one in a different domain than other derivations in the same domain. The definition of second-order generalization given in Chapter 5 needs to be refined so it is useful when changing namespaces.

The other usability issue is how to turn second-order generalizations into useful transforms. The problem is that a set of transforms may be *ambiguous*. For instance, the generalization $\langle[\mathsf{1}] : f\mathsf{a} \rightarrow \mathsf{a},\ [\mathsf{sin}] : f\mathsf{a} \rightarrow \mathsf{sin\ a}\rangle$ leads to the transform $\mathbf{1} \Rightarrow \mathsf{sin}$. Such transforms can be applied anywhere since every subterm contains an instance of $\mathbf{1}$, and so they are not useful. This chapter discusses using contextual information to disambiguate transforms.

The first section of this chapter addresses computing transforms. First, we define a subset to be computed based on those generalizations giving minimal transforms. Second, we give an efficient algorithm to compute preferred generalization terms. Finally, we give an algorithm to compute transforms from the terms.

The second section of this chapter presents heuristics to make the computed transforms useful for replay. First, we discuss further heuristics to select preferred generalizations. Second, we discuss making application explicit to allow generalization across different domains. Third, we describe a method for disambiguating transform rules so that they are useful to replay. Finally, we present the entire algorithm and describe its relationship to the generalization algorithm given in the previous chapter.

This chapter concludes with examples illustrating the use of generalization in replay and a discussion of its limitations.

6.1 Computing Transforms

In this section, we present efficient algorithms to compute a useful subset of $\mathrm{MSC}(a, b)$ and the associated transform rules. We make the following assumptions:

1. The terms a and b are *ground*, *i.e*, they contain no free variables.

2. The terms a and b are first-order, *i.e*, they are combinator terms of type $\mathsf{u} \rightarrow X$.

3. There is a single base type ι; data has the type $\mathsf{u} \rightarrow \iota$ and functions have the type $\iota \cdot \iota \cdots \iota \rightarrow \iota$.

The first two assumptions are possible because all terms are ground in ReFocus and the underlying language is first-order. These two assumptions simplify generalization. Since a and b are first-order terms, they can be written as $t\,(\mathbf{1}_{\mathsf{u}} \mapsto \mathbf{1}_{\mathsf{u}} \cdots \mathbf{1}_{\mathsf{u}})$ (for some grouping of $\mathbf{1}_{\mathsf{u}} \cdots \mathbf{1}_{\mathsf{u}}$) where t does not contain any restructors. Because the restructor τ in **Factor-Restructor** must not have an inverse, and since and λ and ϱ are inverses of δ_{u}, there is no need to consider factoring restructors for first-order terms. The third assumption allows generalizing across distinct domains, such as generalizing a term for computing square roots against a term for reversing a list. Generalizing types can be done using the techniques of [Pfe91], but doing so introduces significant complications. Assuming all types are the same avoids these complications.

The following sections present computing transforms. We define the subset of generalizations to be computed and give algorithms to compute the generalization terms and corresponding sets of transforms. For both algorithms, we briefly analyze their computational complexity and show that they are efficient enough for use in replay.

6.1.1 Limiting Generalizations by Maximizing Term Sizes

An important reason for the presence of multiple generalizations is repeated occurrences of symbols. Each occurrence of a symbol F in one term can be matched against different occurrences of F in another term. For example,

Example 6.1 MSC($\mathsf{FFa}, \mathsf{FFb}$) is

$$\begin{tikzcd}
g_1 : & \mathsf{FFa} & \mathsf{FF}x \arrow[l, "{[\mathsf{a}]}"'] \arrow[r, "{[\mathsf{b}]}"] & \mathsf{FFb} \\
g_2 : & \mathsf{FFa} & h\mathsf{F}x \arrow[l, "{[\mathsf{F}, \mathsf{a}]}"'] \arrow[r, "{[\mathbf{1}, \mathsf{Fb}]}"] & \mathsf{FFb} \\
g_3 : & \mathsf{FFa} & h\mathsf{F}x \arrow[l, "{[\mathbf{1}, \mathsf{Fa}]}"'] \arrow[r, "{[\mathsf{F}, \mathsf{b}]}"] & \mathsf{FFb}
\end{tikzcd}$$

generalize FFa and FFb. Intuitively, g_1 is the best of these three because it reflects the most likely change made by the user: replacing a by b. In other words, g_1 is preferred because more information (that is, a greater number of constants) appears in the generalization term and less information appears in the substitutions. Restricting MSC to generalizations with maximally large source terms leads to simpler sets of transforms between problems. This section present a heuristic used to minimize sets of transforms.

We formalize this heuristic as follows:

Definition 6.2 Let *size* be a metric that determines the "size" of generalizations and $\leq$ be an ordering on sizes. Then,

$$\mathrm{MSC}_{\max}(a, b) = \{g : g \in \mathrm{MSC}(a, b) \text{ and } \forall g' \in \mathrm{MSC}(a, b),\ \mathit{size}(g') \leq \mathit{size}(g)\}$$

An example metric that suits our purpose is *size* : $\mathrm{MSC}(a, b) \rightarrow \langle N, N \rangle$ defined as

$$size(\langle \theta_1 : t \rightarrow a, \theta_2 : t \rightarrow b \rangle) = \langle size(t),\ size(\theta_1) + size(\theta_2) \rangle$$

where for terms t,

$$size(t) = \begin{cases} size(r) + size(s) & \text{if } t = rs \text{ or } t = r \cdot s \\ 0 & \text{if } t \text{ is } p \mapsto q,\ \mathbf{1},\ \text{or a free variable} \\ 1 & \text{otherwise} \end{cases}$$

and for substitutions θ,

$$size(\theta) = \sum_{x \in dom(\theta)} size(\theta(x))$$

For this definition of *size*, we define $\leq$ as

$$\langle a, b \rangle \leq \langle c, d \rangle \iff a < c \text{ or } a = c \text{ and } b \geq d$$

The first component minimizes the size of the generalization term. For example, in 6.1, $\mathrm{MSC}_{\max}(\mathsf{FFa}, \mathsf{FFb})$ contains only g_1 because $size(\mathsf{FF}x)$ is 2 while $size(h\mathsf{F}x)$ is 1. The second component minimizes the size of the resulting transforms. This takes advantage of variables which are bound to the same terms.

Example 6.3 Consider the maximally specific condensed generalizations of $\mathsf{FG} \cdot \mathsf{FH}$ and $\mathsf{GF} \cdot \mathsf{HF}$ in Figure 6.1. Because $size(g_1) = \langle 1, 6 \rangle$, $size(g_2) = \langle 2, 4 \rangle$, and $size(g_3) = \langle 2, 2 \rangle$, g_3 is chosen. The norm of g_3 is the smallest of the three because of the two factored constants and the repeated occurrences of the transforms $\mathsf{F} \Rightarrow \mathbf{1}$ and $\mathbf{1} \Rightarrow \mathsf{F}$.

The size metric given here essentially counts the occurrences of constants. It ignores restructors and symbols such as $\mathbf{1}$ and $\cdot$ because they are artifacts of the representation and counting them would skew the results. This works well on the above examples, but refinements are needed for ReFocus. Suppose the user represents numbers using a successor notation such as $\mathsf{0}$, $\mathsf{S\,0}$, $\mathsf{S\,S\,0}$, and so on. Generalizing $\mathsf{fact} - (\mathsf{S\,S\,0} \cdot \mathsf{0})$ and $\mathsf{fact} + (\mathsf{a} \cdot \mathsf{S\,0})$ (Figure 6.1.1) shows the successor notation leads to anomalies. Since $size(g_a) = \langle 2, 6 \rangle > \langle 1, 10 \rangle = size(g_b)$, counting each S leads to a generalization which reflects similarities between the numbers $\mathsf{S\,S\,0}$ and $\mathsf{S\,0}$ but ignores the common symbol fact. In ReFocus, this is addressed by converting terms representing numbers into numeric constants such as 1 and 2 before applying generalization. The effect is to refine *size* so that $size(\mathsf{0}) = size(\mathsf{S\,S\,0}) = 1$. This method can be applied to other sorts of data structures such as lists and sets. Other possible refinements on *size* include weighting matches

$$\begin{tikzcd}
 & & g(\mathsf{F}h\cdot\mathbf{1}) \arrow[dl, "{[\mathbf{1}\cdot\mathsf{FH},\,\mathsf{G}]}"'] \arrow[dr, "{[\mathsf{G}\cdot\mathsf{HF}]}"] & \\
g_1 : & \mathsf{FG}\cdot\mathsf{FH} & & \mathsf{GF}\cdot\mathsf{HF}
\end{tikzcd}$$

$$\begin{tikzcd}
 & & f\mathsf{F}g\cdot f'\mathsf{F}g' \arrow[dl, "{[\mathbf{1},\mathsf{G},\mathbf{1},\mathsf{H}]}"'] \arrow[dr, "{[\mathsf{G},\mathbf{1},\mathsf{H},\mathbf{1}]}"] & \\
g_2 : & \mathsf{FG}\cdot\mathsf{FH} & & \mathsf{GF}\cdot\mathsf{HF}
\end{tikzcd}$$

$$\begin{tikzcd}
 & & f\mathsf{G}g\cdot f\mathsf{H}g \arrow[dl, "{[\mathsf{F},\mathbf{1}]}"'] \arrow[dr, "{[\mathbf{1},\mathsf{F}]}"] & \\
g_3 : & \mathsf{FG}\cdot\mathsf{FH} & & \mathsf{GF}\cdot\mathsf{HF}
\end{tikzcd}$$

Figure 6.1: Maximally specific condensed generalizations of $\mathsf{FG}\cdot\mathsf{FH}$ and $\mathsf{GF}\cdot\mathsf{HF}$.

$$\begin{tikzcd}
 & & f\,\mathsf{S}\,0 \arrow[dl, "{[-(\mathsf{fact}\,\mathsf{S}\cdot 0)\varrho^{-1}]}"'] \arrow[dr, "{[+(\mathsf{fact}\,\mathsf{a}\cdot\mathbf{1}_X)\lambda^{-1}]}"] & \\
g_a : & -(\mathsf{fact}\,\mathsf{S}\,\mathsf{S}\,0\cdot 0) & & +(\mathsf{fact}\,\mathsf{a}\cdot\mathsf{S}\,0)
\end{tikzcd}$$

$$\begin{tikzcd}
 & & g\,\mathsf{fact}\,x \arrow[dl, "{[-(\mathbf{1}_X\cdot 0)\varrho^{-1},\ \mathsf{S}\,\mathsf{S}\,0]}"'] \arrow[dr, "{[+(\mathbf{1}_X\cdot\mathsf{S}\,\mathsf{S}\,0)\varrho^{-1},\ \mathsf{a}]}"] & \\
g_b : & -(\mathsf{fact}\,\mathsf{S}\,\mathsf{S}\,0\cdot 0) & & +(\mathsf{fact}\,\mathsf{a}\cdot\mathsf{S}\,0)
\end{tikzcd}$$

Figure 6.2: Alternative generalizations of $\mathsf{fact}(\mathsf{S}\,\mathsf{S}\,0) - 0$ and $\mathsf{fact}\,\mathsf{a} + \mathsf{S}\,0$.

by their importance (such as preferring matches between function names over matches between parameter names) or preferring matches between functions with similar-sounding names.

The *size* metric is split into two components for efficiency. While minimizing the size of transforms subsumes maximizing the size of generalization terms, maximizing the size of terms is much simpler. To minimize the size of transforms, we must have the complete substitutions before looking for common transforms; minimizing transforms cannot be done incrementally. In contrast, maximizing generalization terms can be done incrementally. Thus we split the computation of $\mathrm{MSC}_{\mathrm{max}}$ into two parts: maximizing the size of the source term and minimizing the size of transforms. The next section presents computing maximally large generalization terms, and the section which follows presents computing the associated transforms.

6.1.2 Computing Maximally Large Generalizations

The naive way to compute $\mathrm{MSC}_{\mathrm{max}}$ is to repeatedly apply the **Factor-Constant** step of Algorithm 5.46 until no pair of substitutions contains common symbols. Then *size* can be used to find the set of largest generalization terms, and **Merge** applied to find the smallest set of transforms. However, using **Factor-Constant** duplicates work because the order in which the common symbols are chosen often has no effect on the result. This suggests using dynamic programming and matching terms bottom-up. For each pair of terms, we find the match which maximizes the number of matched symbols in the corresponding subterms. This means considering both matching the heads of the terms and matching each term against the other subterms. By recording the results of each match and traversing the terms bottom-up, we avoid comparing subterms more than once. This section describes such an algorithm.

We view generalization is as a form of tree matching, where each term is represented by a *typed tree*:

Definition 6.4 A *typed tree* is a labeled, oriented tree with the restriction that if the node t is labeled by H and H is a function symbol with arity n, then t has n children.

The following definition, inspired by [Pfe91], is used to denote all possible matches between subtrees:

Definition 6.5 A *partial permutation* from n to m is an injective mapping ϕ from S to $\{1 \ldots m\}$ where S is a k-element subset of $\{1 \ldots n\}$ for $k = \min(n, m)$.

Recall that a map f is *injective* if $f(i) = f(j)$ implies $i = j$.

The goal is to find the maximally large embedded typed trees, where

Definition 6.6 Given typed trees s and t, s is *embedded* in t, written

$$s = f(s_1, \ldots, s_m) \trianglelefteq g(t_1, \ldots, t_n) = t$$

if one of the following conditions hold:

i. $s \trianglelefteq t_i$ for some $i \in \{1 \dots n\}$,

ii. $f = g$ (implying $m = n$) and $s_i \trianglelefteq t_i$ for all $i \in \{1 \dots n\}$, or

iii. f denotes a free variable and, for some partial permutation ϕ from n to m, $s_i \trianglelefteq t_{\phi(i)}$ for all $i \in \mathit{dom}\,\phi$.

Given typed trees a and b, we wish to compute

Definition 6.7

$$\begin{aligned} \text{gentrees}(a, b) \quad = \quad & \{t \mid t \trianglelefteq a,\, t \trianglelefteq b, \text{ and for any } t' \text{ such that } t' \trianglelefteq a, t' \trianglelefteq b, \\ & \mathit{size}(t') \leq \mathit{size}(t)\} \end{aligned}$$

Before presenting an algorithm for computing the gentrees function, we present an algorithm which determines the size of the maximally embedded tree. The algorithm to generate the trees themselves is a modified version of this algorithm.

Given two typed trees t_1 and t_2, we index each node of t_1 and t_2 using a preorder traversal with the indices of the roots set to 1. Note that if node p is closer to the root than q, then p's index is smaller than q's. The index of node p is denoted $\mathit{index}(p)$, and the node (and subtree) of t with index i is denoted t/i. The label of node p is denoted $\mathit{label}(p)$. When matching t_1/i to t_2/j, there are four possibilities to consider:

1. matching t_1/i to some subtree of t_2/j,

2. matching t_2/j to some subtree of t_1/i,

3. matching some permutation of the subtrees of t_1/i to the subtrees of t_2/j, and

4. if $\mathit{label}(t_1/i) = \mathit{label}(t_2/j)$, matching each subtree in parallel.[1]

Let $|t|$ be the number of nodes in tree t. To make these sets of possible matches explicit, we define two $|t_1| \times |t_2|$ matrices N and N' such that N contains the matches in items 1–3 and N' contains the matches in item 4. Thus $N_{i,j} \cup N'_{i,j}$ represents the set of all possible matches between nodes t_1/i and t_2/j.

Definition 6.8 Let the children of t_1/i be $\overline{p_m}$ and the children of t_2/j be $\overline{q_n}$ (for $m, n \geq 0$) where $\overline{u_n}$ is the sequence $u_1, u_2, \dots u_n$. Then $N_{i,j}$ is the set containing the sets

- $\{\langle i, \mathit{index}(q)\rangle \mid q \in \overline{q_m}\}$

[1]Since t_1 and t_2 are typed trees, t_1/i and t_2/j contain the same number of subtrees if their labels are the same.

- $\{\langle index(p), j\rangle \mid p \in \overline{p_n}\}$
- $\{\langle index(p_k), index(q_{\phi(k)})\rangle \mid \phi$ is a partial permutation from n to m and $k \in dom\ \phi\}$

and

$$N'_{i,j} = \begin{cases} \{\langle i,j\rangle,\ \langle index(p_k), index(q_k)\rangle \mid 1 \le a \le n\} & \text{if } label(t_1/i) = label(t_2/j) \\ \emptyset & \text{otherwise} \end{cases}$$

That is, $N'_{i,j}$ contains the matches to be made if the roots are the same, and $N_{i,j}$ is the set of matches obtained by ignoring any possible match between the roots. Note that $N'_{i,j}$ is a set of node-index pairs while $N_{i,j}$ is a *set of sets* of node-index pairs.

Using N and N', we define two $|t_1| \times |t_2|$ matrices M and M' such that $M_{i,j}$ is the size of the maximally embedded tree obtained from $N_{i,j}$ and $M'_{i,j}$ is the size obtained from $N'_{i,j}$. If P is a set of node-index pairs, let $M_{\Sigma P}$ denote

$$\sum_{\langle p,q\rangle \in P} M_{p,q}$$

Then M and M' are mutually defined as follows:

Definition 6.9

$$\begin{aligned} M'_{i,j} &= \begin{cases} 0 & \text{if } N'_{i,j} = \emptyset \\ 1 + M_{\Sigma(N'_{i,j} \setminus \langle i,j\rangle)} & \text{otherwise} \end{cases} \\ M_{i,j} &= \max\{M'_{i,j}, M_{\Sigma P} \mid P \in N_{i,j}\} \end{aligned}$$

That is, if $N'_{i,j}$ is non-empty, then $M'_{i,j}$ is the result of matching the roots and corresponding subtrees of t_1/i and t_2/j. In turn, $M_{i,j}$ is the maximum of comparing the roots and comparing all possible combinations of subtrees. Since N and N' give all ways of matching any pair of subtrees,

$$M_{1,1} = |t| \text{ such that } t \in \text{gentrees}(a,b)$$

That is, $M_{1,1}$ is the size of each maximally large embedded typed tree. Since $i' > i$ implies t/i' is a subtree of t/i, $M_{i,j}$ and $M'_{i,j}$ depend only on $M_{i',j'}$ for $i' > i$ and $j' > j$. Thus M and M' can be computed by the nested loops

```
for i = $n$ downto 1
    for j = $m$ downto 1
        compute $M'_{i,j}$ and $M_{i,j}$
```

In this way, dynamic programming is used to compute the size of the maximally large embedded typed tree. (Note that N and N' are used only to simplify the description of the algorithm; an implementation need not compute them explicitly.)

In the worst case, computing M takes exponential time: If the maximum out-degree of any node in either tree is k, then computing $M_{i,j}$ involves $1 + |N_{i,j}| \leq 1 + k + k + k!$ comparisons. Thus if the size of both trees (terms) is n, then the algorithm takes $O(k!n^2)$ time.[2] Since MSC_{max} may contain an exponential number of generalizations (*cf.* Example 5.41), there is no algorithm which finds the maximally large trees in polynomial time. However, k is normally a small constant, so computing M can be thought of as taking quadratic time.

To find the maximally large embedded trees, we define a matrix I, parallel to M, which contains the combinations of node indices comprising the maximally large subtrees. That is,

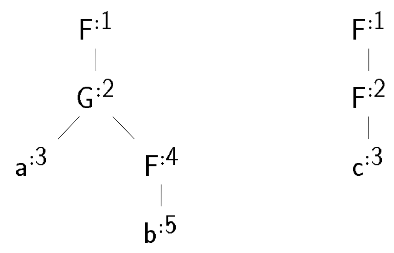
Definition 6.10

$$I_{i,j} = \{P \in N_{i,j} \mid M_{i,j} = M_{\Sigma P}\} \cup \{N'_{i,j} \mid M_{i,j} = M'_{i,j}\}$$

The first subset (based on N) gives the matches between subtrees. The second subset (based on N') gives the match between the roots and subtrees in parallel. In both subsets, M is used to ensure that the corresponding matches are maximally large. Thus $I_{1,1}$ is the root of a directed, acyclic graph containing all possible combinations of indices used to make the maximally large common tree.

Constructing the set of maximal trees is done by traversing all paths of this graph, choosing some index-pair set P from $I_{i,j}$, and including the label of t_1/i (which is the same as $label(t_2/j)$) if P contains the pair $\langle i, j\rangle$. These trees can then be used to generate the terms by introducing free variables at the appropriate places. However, computing the terms is not necessary because I gives enough information to generate the sets of transformation rules. This is described further in Section 6.1.3. But first, we illustrate computing M and I:

Example 6.11 Let $s = \mathsf{F\,G(a \cdot F\,b)}$ and $t = \mathsf{F\,F\,c}$. The indexed trees are



[2]This assumes that the $k!$ possible permutations are compared in less than the $O(k!\,k)$ operations needed to generate all possible lists and sum up each individually. There are a number of solutions; the implementation presented in this thesis uses [Der75] to generate the permutations by swapping adjacent elements in an array. This gives an algorithm with $O(k + k!)$ additions and comparisons.

where the $:i$ represent the node indices, and M is

		t's indices $\rightarrow$		
		1	2	3
s's	1	2	1	0
indices	2	1	1	0
$\downarrow$	3	0	0	0
	4	1	1	0
	5	0	0	0

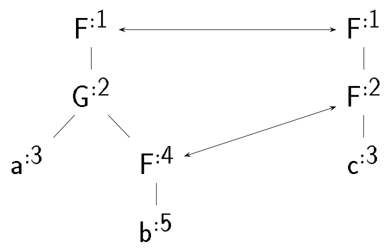
For example,

$$M_{\langle 1,1\rangle} = \max\{1 + M_{\Sigma\{\langle 2,2\rangle\}}, M_{\Sigma\{\langle 1,2\rangle\}}, M_{\Sigma\{\langle 2,1\rangle\}}\} = \max\{1+1, 1, 1\} = 2$$

Matrix I is (where the indices are the same as in the previous table)

	1	2	3
1	$\{\{\langle 1,1\rangle, \langle 2,2\rangle\}\}$	$\{\{\langle 1,2\rangle, \langle 2,3\rangle\}, \{\langle 2,2\rangle\}\}$	$\emptyset$
2	$\{\{\langle 4,2\rangle\}, \{\langle 2,2\rangle\}, \{\langle 4,1\rangle\}\}$	$\{\{\langle 4,2\rangle\}\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$
4	$\{\{\langle 4,1\rangle, \langle 5,2\rangle\}, \{\langle 4,2\rangle\}\}$	$\{\{\langle 4,2\rangle, \langle 5,3\rangle\}\}$	$\emptyset$
5	$\emptyset$	$\emptyset$	$\emptyset$

The maximally embedded tree is then composed of the node pairs $\{\langle 1,1\rangle, \langle 4,2\rangle\}$:



This represents the generalization term $\mathsf{F}\,x\,\mathsf{F}\,y$. Note that $N_{1,1}$ includes $\{\langle 1,2\rangle\}$ and $\{\langle 2,1\rangle\}$, but $I_{1,1}$ does not since these lead to maximally embedded trees of size 1.

This section has described how to compute maximally large generalization terms. But since I contains enough information to construct both the terms and substitutions, we can also use it to compute $\mathrm{MSC}_{\mathrm{max}}$. However, ReFocus does not need the full generalizations; all it needs is the resulting transform rules. Computing sets of transforms is described in the next section.

6.1.3 Computing Sets of Transforms

Given I, we can compute the sets of transforms by constructing each maximally large generalization term, finding the appropriate substitutions, and pairing the bindings in the substitutions. However, computing the maximally large generalization terms from I is complex, and second-order matching takes exponential time in the worst case.[3] Because of this complexity and expense, a much better algorithm can be obtained by computing the transforms directly from I. This section presents such an algorithm.

Given terms a and b, let their indexed trees be a' and b' and let I be as specified by Definition 6.10. Using depth-first search through the graph embedded in I (*cf.* [RND77]), we can find all sets of consistent matches between the nodes of a' and b' by collecting those node-index pairs $\langle i,j\rangle$ which occur in $I_{i,j}$. We denote such a set by $\mathrm{matches}_{\langle i,j\rangle}$:

Definition 6.12 Let P be such that $P \in I_{i,j}$. Then

$$\mathrm{matches}_{\langle i,j\rangle} = \bigcup_{\langle p,q\rangle \in P \setminus \langle i,j\rangle} \mathrm{matches}_{\langle p,q\rangle} \cup \begin{cases} \{\langle i,j\rangle\} & \text{if } \langle i,j\rangle \in P \\ \emptyset & \text{otherwise} \end{cases}$$

The set $\mathrm{matches}_{\langle i,j\rangle}$ represents some set of matches between nodes with identical labels between the trees a'/i and b'/j. Thus $\mathrm{matches}_{\langle 1,1\rangle}$ is a set of consistent matches between the labels of a' and b'. Note that there is often more than one set satisfying the definition of matches. We define the set of *all* maximal matches as

Definition 6.13

$$\mathcal{M} = \{\mathrm{matches}_{\langle 1,1\rangle}\}$$

Let $\mathcal{P} \in \mathcal{M}$ and let ϕ be a bijection from P to a set of fresh (first-order) variables. Each $\langle p,q\rangle \in P$ is used to make a transform rule by traversing the subtrees of a'/p and b'/q, including all nodes in the rule except those with indices appearing in $\mathcal{P}$. We give two functions to compute the transforms; one to compute the left-hand side and another to compute the right-hand side. These will then be combined to construct the transforms.

Algorithm 6.14 If the children of a'/i are $k_1 \ldots k_n$, then let

$$\mathrm{L}_i = \begin{cases} \phi\langle i,j\rangle & \text{if } \exists j.\langle i,j\rangle \in \mathcal{P} \\ \mathit{label}(a'/i)(\mathrm{L}_{k_1} \cdot \mathrm{L}_{k_2} \cdots \mathrm{L}_{k_n}) & \text{otherwise} \end{cases}$$

[3] For instance, there are $n!$ matches between $f(\mathtt{F}x_1 \cdot \mathtt{F}x_2 \cdots \mathtt{F}x_n)$ and $\mathtt{Fa}_1 \cdot \mathtt{Fa}_2 \cdots \mathtt{Fa}_n$.

This gives the left-hand side of transform rules. Likewise, let the children of b'/j be $l_1 \ldots l_n$, and let

$$\mathrm{R}_j = \begin{cases} \phi\langle i,j\rangle & \text{if } \exists i.\langle i,j\rangle \in \mathcal{P} \\ \mathit{label}(b'/j)(\mathrm{R}_{l_1} \cdot \mathrm{R}_{l_2} \cdots \mathrm{R}_{l_n}) & \text{otherwise} \end{cases}$$

The difference between L and R is that L matches on the left components in $\mathcal{P}$ while R matches on the right components. Given a pair of matching nodes $\langle i,j\rangle$, we compute the transforms from the children. Let the children of a'/i be $k_1 \ldots k_n$ and the children of b'/j be $l_1 \ldots l_n$. Then

$$\mathit{transforms}_{\langle i,j\rangle} = \bigcup_{x \in \{1..n\}} \mathrm{L}_{k_x} \Rightarrow \mathrm{R}_{l_x}$$

and the set of transforms associated with $\mathcal{P}$ is given by

$$\{\mathrm{L}_1 \Rightarrow \mathrm{R}_1\} \cup \bigcup_{\langle i,j\rangle \in P} \mathit{transforms}_{\langle i,j\rangle}$$

Finally, we remove all transforms of the form $x \Rightarrow x$ from this set. These transforms arise when all of the children of a'/i and b'/j match for some $\langle i,j\rangle \in \mathcal{P}$ or when the top two nodes match.

Example 6.15 The set $\text{matches}_{\langle 1,1\rangle}$ for Example 6.11 is

$$\{\langle 1,1\rangle\} \cup \text{matches}_{\langle 2,2\rangle} = \{\langle 1,1\rangle\, \langle 4,2\rangle\}$$

If $\phi\langle 1,1\rangle = x$ and $\phi\langle 4,2\rangle = y$, then the computed transformation rules are

$$\{\mathrm{L}_1 \Rightarrow \mathrm{R}_1,\, \mathrm{L}_2 \Rightarrow \mathrm{R}_2,\, \mathrm{L}_5 \Rightarrow \mathrm{R}_3\} \quad = \quad \{x \Rightarrow x,\, \mathsf{G}(\mathsf{a} \cdot y) \Rightarrow y,\, \mathsf{b} \Rightarrow \mathsf{c}\}$$

The trivial rule $x \Rightarrow x$ is removed.

Thus computing the set of transforms from I is done in three steps:

1. Find all sets of consistent matches between nodes corresponding to the maximally embedded trees.

2. Construct transforms from each set.

3. Delete duplicate transforms.

These steps do not add to the time complexity of generalization. The first step is proportional to the number of maximal generalizations; this can be exponential in the worst case, but it is normally much less. In the second step, if we store the nodes in a vector to allow simple tests for membership in $\mathcal{P}$, generating the transforms for each $\mathcal{P} \in \mathcal{M}$ takes time proportional to twice the number of symbols in a and b. Thus the dominant expense in generalization is computing the maximally large generalization terms.

We next consider heuristics to increase the usefulness of the produced transforms.

6.2 Heuristics for Improving Transforms

While the previous section gives an efficient algorithm for computing generalizations, we are also concerned about usefulness. There are three issues to be addressed:

- reducing the number of alternative generalizations for use by replay,
- making function application explicit to improve generalization across distinct domains, and
- incorporating contextual information in rules to make them more specific.

We consider each of these in turn, and conclude with a description of the complete generalization algorithm used in ReFocus and a comparison between it and the algorithm in Chapter 5.

6.2.1 Using Historical and User-supplied Information

While $\mathrm{MSC}_{\max}(a,b)$ is usually much smaller than $\mathrm{MSC}(a,b)$, the following example shows that it still may contain multiple elements:

Example 6.16 Let g_- be the generalization

$$\begin{tikzcd}
& f\,\mathsf{cos}\,g(\mathsf{a}\cdot\mathsf{b}) \arrow[dl, "{[\mathsf{F},+]}"'] \arrow[dr, "{[\mathsf{G}(\mathbf{1}_\iota\cdot\mathsf{cos}\,*(\mathsf{a}\cdot\mathsf{b}))\varrho^{-1},-]}"] & \\
\mathsf{F}\,\mathsf{cos}\,+(\mathsf{a}\cdot\mathsf{b}) & & \mathsf{G}(\mathsf{cos}\,-(\mathsf{a}\cdot\mathsf{b})\cdot\mathsf{cos}\,*(\mathsf{a}\cdot\mathsf{b}))
\end{tikzcd}$$

Note this is based on matching $+$ to $-$. We can also match $+$ to $*$, giving g_*:

$$\begin{tikzcd}
& f\,\mathsf{cos}\,g(\mathsf{a}\cdot\mathsf{b}) \arrow[dl, "{[\mathsf{F},+]}"'] \arrow[dr, "{[\mathsf{G}(\mathsf{cos}\,-(\mathsf{a}\cdot\mathsf{b})\cdot\mathbf{1}_\iota)\lambda^{-1},*]}"] & \\
\mathsf{F}\,\mathsf{cos}\,+(\mathsf{a}\cdot\mathsf{b}) & & \mathsf{G}(\mathsf{cos}\,-(\mathsf{a}\cdot\mathsf{b})\cdot\mathsf{cos}\,*(\mathsf{a}\cdot\mathsf{b}))
\end{tikzcd}$$

The sizes of these are the same: $\langle 3,8\rangle$.

This section considers using semantic information to further reduce the number of alternative generalizations.

In particular, we consider using user-supplied and historical information about name correspondences. For example, g_- might be preferred because of the user's bias towards matching $+$ to $-$ instead of $*$. Allowing the user to express this bias can be particularly helpful when the user is using replay for derivation-by-analogy since then it is likely that the two problems will have entirely different sets of symbols. Alternatively, generalization g_* might be preferred because previous uses of replay were successful when generalization matched $+$ to $*$. This sort of bias can be helpful when using replay for change propagation. This section considers both sorts of biases.

Historical and user-supplied information can take a number of forms. One is to store statistical information about previous replays to guide future ones. When replay has been successful, the system would store the correspondences that were used. During replay, generalization would favor correspondences that have occurred more frequently. A second method is for the user to supply a database specifying the semantic distance between symbols (*cf.* [MH91]). A third approach is to ask the user to choose between alternatives.

Each of these methods has weaknesses. Asking the user to choose a best match makes replay less automatic, possibly to the point where it is easier to repeat a derivation by hand. A user-supplied database can become a maintenance problem in itself, though it might be worthwhile if a large number of derivations are in a specific domain. Storing statistical information has the problem that matches that work well for one example may work poorly for another. Experimentation is needed to determine if any one of these, or some combination, works best in practice; this is left as future work.

Of these two biases, only historical information is used by ReFocus. As ReFocus generates transforms, it stores them in the derivation tree so that they are available to later replay operations. When ReFocus needs to find the closest match between alternative terms, it applies the stored transforms to each pair of terms before generalizing them. This increases the consistency between generalizations in a very simple way. It is especially important in derivation-by-analogy, as illustrated in Example 6.22 (page 109). However, while this method works for many examples, it is not robust. It depends strongly on having constructed the right set of transforms in previous replay operations. In particular, we have found that when generalization is used to construct new transforms, applying transforms to the terms being generalized often makes the new transforms less accurate. More work on incorporating historical information is needed.

6.2.2 Representing Terms

A second usability issue to be addressed is ensuring that generalization produces useful results when terms come from distinct domains. This section discusses making function application explicit to improve replaying derivations when the symbols are very different.

Unless the representation is changed, generalization will fail to produce useful results in simple cases. For example, suppose the user has a proof for the associativity of append:

$$\mathsf{append}(\mathsf{a} \cdot \mathsf{append}(\mathsf{b} \cdot \mathsf{c})) = \mathsf{append}(\mathsf{append}(\mathsf{a} \cdot \mathsf{b}) \cdot \mathsf{c})$$

Then suppose the user changes the notation so that append is denoted using the infix operator $+\!\!+$:

$$\mathsf{u} +\!\!+ (\mathsf{v} +\!\!+ \mathsf{w}) = (\mathsf{u} +\!\!+ \mathsf{v}) +\!\!+ \mathsf{w}$$

Given the constraints for condensed generalizations, the generalization term for these two equations is trivial: $x = y$. The resulting generalization is not useful to ReFocus.

A solution is to use a notation that is commonly used in first-order generalization [Owe90]: represent terms so that application is explicit, such as

$$@_2(\mathsf{append} \cdot \mathsf{a} \cdot @_2(\mathsf{append} \cdot \mathsf{b} \cdot \mathsf{c})) = @_2(\mathsf{append} \cdot @_2(\mathsf{append} \cdot \mathsf{a} \cdot \mathsf{b}) \cdot \mathsf{c}) \tag{6.1}$$

and

$$@_2(+\!\!+ \cdot \mathsf{u} \cdot @_2(+\!\!+ \cdot \mathsf{v} \cdot \mathsf{w})) = @_2(+\!\!+ \cdot @_2(+\!\!+ \cdot \mathsf{u} \cdot \mathsf{v}) \cdot \mathsf{w}) \tag{6.2}$$

where $@_n(f \cdot \overline{u_n})$ represents applying f to $\overline{u_n}$.[4,5] By treating $@_n$ operators as other constants during generalization, we capture common structures and obtain transformation rules from one namespace to another. For example, we can match append to $+\!\!+$ and the various variable names to the corresponding positions:

$$\begin{tikzcd} & @_2(f \cdot x \cdot @_2(f \cdot y \cdot z)) = @_2(@_2(f \cdot @_2(f \cdot x \cdot y) \cdot z)) \arrow[dl, "{[\mathsf{append}, \mathsf{a}, \mathsf{b}, \mathsf{c}]}"'] \arrow[dr, "{[+\!\!+, \mathsf{u}, \mathsf{v}, \mathsf{w}]}"] & \\ (6.1) & & (6.2) \end{tikzcd}$$

Note that the multiple occurrences of f in the source term captures the consistent renaming of append to $+\!\!+$. This happens because the $@_n$ notation allows introducing multiple variables in cases where the definition of MSC allows only one variable.

[4] For clarity, the equality operator is shown using infix notation. In ReFocus, equality is treated specially so that it always appears in the generalization.

[5] As in Chapter 5, we use $\overline{x_n}$ to denote $x_1 \cdot x_2 \cdots x_n$. As mentioned earlier, we also use it to represent $x_1, x_2, \ldots x_n$; the context makes it clear which is meant.

One consequence of using the $@_n$ notation is that it treats function symbols as if they have the type $\mathsf{u} \to \iota$. However, the following observation shows that using $@_n$ will not result in a transform mapping a function symbol to a first-order term or *vice versa*:

Observation 6.17 Given typed trees s and t with the respective children $\overline{p_m}$ and $\overline{q_n}$, a transform is computed from p_i and q_j if and only if $label(s) = label(t)$ and $i = j$.

This follows directly from the definition of *transforms* in Algorithm 6.14. Since function symbols appear only in the context of an $@_n$ operator in the above notation, both sides of a transform $r \Rightarrow r'$ must be function symbols or both must be first-order terms (in the object language).

Using $@_n$ to make application allows generalization to obtain useful results when the instance terms are from distinct domains. In effect, it counteracts the restriction in MSC against redundant variables without introducing too many generalizations. An alternative solution is to loosen the restriction on adjacent variables as discussed at the end of Section 5.3.3; investigating this solution is left as future work. In the remainder of this chapter, we assume the terms being generalized have been converted into the $@_n$ notation.

6.2.3 Incorporating Context

By design, the transforms obtained from generalization contain as few symbols as possible. However, some cases require more information. Suppose a set of rules transform a to b. Transforms which delete or rename symbols in a are relatively easy to use, but transforms which simply introduce symbols in b are not because they can be applied almost anywhere. Furthermore, symbols may occur in different contexts in a, and different transforms may apply to different occurrences of the symbols. Thus contextual information is often needed to disambiguate transforms.

There are two parts to disambiguating transforms: identifying when a transform is ambiguous and determining what contextual information to add. Let $\langle \theta_1 : t \to a, \theta_2 : t \to b \rangle$ be a generalization of a and b, and let $x, y \in dom(\theta_1)$. One criteria for deciding that a transform $\theta_1(x) \Rightarrow \theta_2(x)$ is ambiguous is if $\theta_1(x)$ is an identity or $\theta_1(x) = \theta_1(y)$ for some $y \neq x$. In ReFocus, such transforms are disambiguated by incorporating the context of $\theta_1(x)$ from a. Let a be $a[@_n(H \cdot \overline{a_n})]$ where $\theta_1(x)$ corresponds to $a_k \in \overline{a_n}$. Then in place of the rule $\theta_1(x) \Rightarrow \theta_2(x)$, we use

$$@_n(H, z_1, \ldots, z_{k-1}, \theta_1(x), \ldots, z_n) \Rightarrow @_n(H, z_1, \ldots, z_{k-1}, \theta_2(x), \ldots, z_n)$$

where each z_i is a free variable.

Example 6.18 Consider the generalization

$$\begin{tikzcd} @_1(\mathsf{F} \cdot \mathsf{A}) & @_1(\mathsf{F} \cdot h\,\mathsf{A}) \arrow[l, "{[\mathbf{1}]}"'] \arrow[r, "{[@_1(\mathsf{G} \cdot \mathbf{1}_X)\lambda^{-1}]}"] & @_1(\mathsf{F} \cdot @_1(\mathsf{G} \cdot \mathsf{A})) \end{tikzcd}$$

Instead of the transform $x \Rightarrow @_1(\mathsf{G} \cdot x)$, we use $@_1(\mathsf{H} \cdot x) \Rightarrow @_1(\mathsf{H} \cdot @_1(\mathsf{G} \cdot x))$.

A second reason for considering a transform to be ambiguous is if it is of the form $@_n(x \cdot \overline{u_n}) \Rightarrow @_n(x \cdot \overline{v_n})$, where x is a variable. When the function applications are converted back to implicit application, this gives the transform $x(\overline{u_n}) \Rightarrow x(\overline{v_n})$. Experience has shown this usually makes the application of the rule ambiguous because it is rare that the arguments to *all* functions in a derivation should be transformed in the same way. Furthermore, even if $\overline{u_n} = \mathsf{Q}$ where Q is a constant that only appears in limited contexts, the flexibility of second-order matching means that $x\,\mathsf{Q}$ can match almost any subterm containing Q, not just those of the form $\mathsf{F}\,\mathsf{Q}$. Our solution is to identify the function symbol which x abstracts and substitute the symbol for each occurrence of x in the rule.

Example 6.19 Consider the generalization

$$@_2(\mathsf{F} \cdot (\mathsf{A} \cdot \mathsf{B})) \xleftarrow{[@_2]} f(\mathsf{F} \cdot (\mathsf{A} \cdot \mathsf{B})) \xrightarrow{[@_2(\mathbf{1}_X \cdot \gamma)]} @_2(\mathsf{F} \cdot (\mathsf{B} \cdot \mathsf{A}))$$

Instead of the transform $@_2(x_1, x_2, x_3) \Rightarrow @_2(x_1, x_3, x_2)$, we use $@_2(\mathsf{H}, x_2, x_3) \Rightarrow @_2(\mathsf{H}, x_3, x_2)$.

These methods could be refined in a number of ways. Other criteria, such as comparing new rules to previously generated rules, could be used to determine when a rule is ambiguous. More contextual information could be included in rules to make them more specific. Also, type information could used both to determine when a rule is ambiguous and how to disambiguate it. Such refinements are left as future work.

6.2.4 Computing Transforms from Instance Terms

This section has presented several ways to improve the transform rules computed in Sections 6.1.2 and 6.1.3. These include reducing the number of generalizations by examining information not available to generalization, changing representations for increased flexibility, and adding information to rules to make them more specific to particular parts of specifications. This section describes the complete generalization algorithm used in ReFocus and compares it to the algorithm in Chapter 5.

The following sequence of steps is used in ReFocus to compute sets of transforms from instance terms:

Algorithm 6.20

1. Insert the $@_n$ notation into terms as described in Section 6.2.2.

2. Convert the terms into labeled trees, indexing each node as described in Section 6.1.2.

3. Compute M, M', and I as described in Section 6.1.2.

4. Compute the transform rules from the indices of the matched nodes as in Algorithm 6.14.

5. Delete rules of the form $x \Rightarrow x$.

6. Add contextual information to ambiguous rules.

7. Remove the $@_n$ operators from the transform rules.

8. Delete any duplicate transforms.

9. Find the smallest sets of transforms.

If more than one set of transforms remains, ReFocus assumes that they all are equally adequate and selects one at random.

When generalization is used to pick the closest match (such as in Example 5.51), we make the following changes:

- As described in Section 6.2.1, transforms from previous generalizations are applied to the terms before step 1.

- Step 6 is skipped so that contextual information does not skew the results.

- Step 7 is skipped so that structural matches—in the form of matching $@_n$ operators—are considered when deciding which match is best.

It is useful to compare Algorithm 6.20 to 5.46. Algorithm 5.46 contains four steps: **Delete**, **Merge**, **Factor-Constant**, and **Factor-Restructor**. Step 3, computing I, corresponds to repeatedly applying **Factor-Constant** until as many common symbols appear in the generalization term as possible and then selecting the generalizations which maximize the size of the generalization term. Step 5, removing trivial transforms, corresponds to applying **Delete**.[6] Since we assume all terms are first-order, there is no need to apply **Factor-Restructor**. Thus Algorithm 6.20 corresponds to the rule application sequence

$$\textbf{Factor-Constant}^*;\ \textbf{Delete}^*;\ \textbf{Merge}^*$$

where * denotes applying a rule until it cannot be applied any more. This is followed by choosing generalizations which lead to minimally large transforms. Since **Delete** does not affect **Factor-Constant** and **Merge**, the rules of Algorithm 5.46 can be applied as

$$(\textbf{Factor-Constant} \mid \textbf{Merge})^*;\ \textbf{Delete}^*$$

[6]The proof of correctness for Algorithm 5.46 (in Section B.3) shows that when the terms are first-order, **Delete** is essentially used to remove substitutions of the form $f \mapsto \mathbf{1}$. These substitutions give rise to the transforms $x \Rightarrow x$.

where | denotes applying either of two rules. Thus for the terms we are interested in, the primary difference between Algorithms 6.20 and 5.46 is that **Merge** steps are not intermingled with **Factor-Constant** steps.

The next section illustrates using Algorithm 6.20 in replay problems.

6.3 Examples

This section shows the use of second-order generalization in replay. All of these examples have been tested using an implementation of ReFocus. The first example illustrates using generalization to propagate changes in specifications. This emphasizes using transforms to update terms. The second illustrates using generalization to support derivation-by-analogy. This emphasizes using analogy to find which sets of alternatives provide the best match. The third example shows some of the limitations of second-order generalization and suggests when it is likely to fail.

In the following examples, we use standard (non-combinator) notation for the terms in Focus derivations and use combinator notation when considering generalization. To reduce the number of parentheses, we assume that $\cdot$ associates to the right. While generalization is done using the $@_n$ notation, we omit $@_n$ in the text for clarity. In each case, the full generalizations including $@_n$ operators are given in figures. As a result of removing the $@_n$ notation, some generalizations appear to contain adjacent variables; these variables are not adjacent in the $@_n$ versions.

As in Section 5.6, we compare matches between generalizations by comparing norms as defined in 5.52. For $|t|$, we use $size(t)$ as defined in Section 6.1.1 with the modification that the size of a term representing a numeric value (such as 0, S0, *etc.*) is 1. The $@_n$ operators are treated as constant function symbols and so are included in $size(t)$.

Example 6.21 The predicate sorted tests if a list is sorted from low to high by checking that each entry in the list smaller than all the remaining items in the list:

```
sorted(Nil)      → True
sorted(a.l)      → smaller(l, a) & sorted(l)

smaller(Nil, x) → True
smaller(y.l, x)  → x < y & smaller(l, x)
```

This predicate makes $O(n^2)$ comparisons (where n is the size of the list). Figure 6.3 outlines a derivation of a linear-time version of the predicate sortedp. The key is proving the property

```
smaller(l, a) {a < y & sortedp(y.l)} = True
```

```
Focus: sortedp(x) = sorted(x)
[closed with program:]
sortedp(Nil) → True
sortedp(a.Nil) → sortedp(Nil)
sortedp(a.y.l1) → a < y & sortedp(y.l1)

  cases from sorted(x):

  1. case x == Nil
     sortedp(Nil) = True
     sortedp(Nil) = True

  2. case x == a.l
     sortedp(a.l) = smaller(l, a) & sorted(l)
     sortedp(a.l) = smaller(l, a) & sortedp(l)

       cases from smaller(l, a):

       1. case l == Nil
          sortedp(a.Nil) = True & sortedp(Nil)
          sortedp(a.Nil) = sortedp(Nil)

       2. case l == y.l1
          sortedp(a.y.l1) = a < y & smaller(l1, a) & sortedp(y.l1)
          sortedp(a.y.l1) = a < y & sortedp(y.l1)


Prove: smaller(l, a) {a < y & sortedp(y.l)} = True
[closed with properties:]
smaller(l, a) {sortedp(y.l) & a < y} = True

  cases from smaller(l, a): ...

       cases from sortedp(y.l1): ...

       [... remainder of proof omitted ...]
```

Figure 6.3: Deriving a linear-time sorted predicate.

which states that if a is smaller than some y and the list y.l is sorted, then a is smaller than l.[7] The property is used to remove the redundant test smaller in case 2.2

sortedp(a.y.l1) = a < y & smaller(l1, a) & sortedp(y.l1)

to obtain

sortedp(a.y.l1) = a < y & sortedp(y.l1)

The proof of the property involves several expansions and is omitted for brevity.

Suppose that the user changes the specification of sorted in two ways:

- The arguments to smaller are swapped so they are in a more natural order with the smaller value as the first argument.
- The < function is generalized to allow sorting values from low to high or high to low by specifying whether the list goes Up or Down.

The new specification is

```
sorted(Nil, dir)       → True
sorted(a.l, dir)       → smaller(a, l, dir) & sorted(l, dir)

smaller(x, Nil, dir)   → True
smaller(x, y.l, dir)   → less(x, y, dir) & smaller(x, l, dir)

less(a, b, Up)         → a < b
less(a, b, Down)       → b < a
```

Note that the new versions of the functions must support the extra parameter specifying the direction of the sort. When the user edits the node containing the function definitions, Focus constructs analogical maps by comparing the specifications for each case. Generalization is used to find the closest matches between cases. For example, matching

$$\mathsf{smaller}(.(\mathsf{y}\cdot\mathsf{l})\cdot\mathsf{x}) \rightarrow \&(<(\mathsf{x}\cdot\mathsf{y})\cdot\mathsf{smaller}(\mathsf{l}\cdot\mathsf{x}))$$

against the new definition of smaller,

i. $\mathsf{smaller}(\mathsf{x}\cdot\mathsf{Nil}\cdot\mathsf{dir}) \rightarrow \mathsf{True}$

ii. $\mathsf{smaller}(\mathsf{x}\cdot .(\mathsf{y}\cdot\mathsf{l})\cdot\mathsf{dir}) \rightarrow \&(\mathsf{less}(\mathsf{x}\cdot\mathsf{y}\cdot\mathsf{dir})\cdot\mathsf{smaller}(\mathsf{x}\cdot\mathsf{l}\cdot\mathsf{dir}))$

gives the terms and transforms

[7]This property is expressed as an equality rather than a rewrite-rule since it cannot be oriented due to the variable y in the condition. It is applied by explicitly rewriting a subterm with `smaller` at the topmost position.

i. $w_1(\mathsf{smaller} \cdot \mathsf{x}) \rightarrow w_2$

$\{x_1(.(\mathsf{y} \cdot \mathsf{l}) \cdot x_2) \Rightarrow x_1(x_2 \cdot \mathsf{Nil} \cdot \mathsf{dir}), \&(< (\mathsf{x} \cdot \mathsf{y}) \cdot \mathsf{smaller}(\mathsf{l} \cdot \mathsf{x})) \Rightarrow \mathsf{True}\}$

ii. $w_1(\mathsf{smaller} \cdot .(\mathsf{y} \cdot \mathsf{l}) \cdot \mathsf{x}) \rightarrow \&(w_2(\mathsf{x} \cdot \mathsf{y}) \cdot w_1(\mathsf{smaller} \cdot \mathsf{l} \cdot \mathsf{x}))$

$\{x_1(x_2 \cdot x_3) \Rightarrow x_1(x_3 \cdot x_2 \cdot \mathsf{dir}), < (x_1 \cdot x_2) \Rightarrow \mathsf{less}(x_1 \cdot x_2 \cdot \mathsf{dir})\}$

For the detailed versions of the generalizations, see Figure 6.4. The norm of the first case is $19/(18+7) = 76\%$, while the norm of the second case is just $8/(18+21) \approx 21\%$. This difference arises because virtually none of the constants are in the generalization term for the first case. Thus we choose the second set of transforms, add inner contextual information, and remove the $@_n$ operators to get

$$\{\mathsf{smaller}(u, v) \Rightarrow \mathsf{smaller}(v, u, \mathsf{dir}), u < v \Rightarrow \mathsf{less}(u, v, \mathsf{dir})\}$$

The transform for $<$ introduces dir, while the transform for smaller both introduces dir and reverses the arguments. Using similar methods to compare the definition of sorted adds the transform

$$\mathsf{sorted}(u) \Rightarrow \mathsf{sorted}(u, \mathsf{dir})$$

To derive the new version of sortedp, we use the focus specification

sortedp(x, dir) = sorted(x, dir)

Comparing this specification against the original gives the transform

$$\mathsf{sortedp}(u) \Rightarrow \mathsf{sortedp}(u, \mathsf{dir})$$

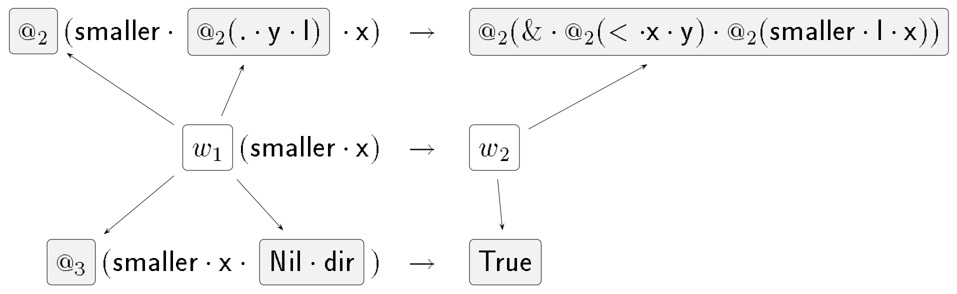
Applying the transforms during replay gives the derivation in Figure 6.5. During replay, ReFocus uses these transforms in primarily two ways. First, they are used to update the terms in the `expand` steps:

$$\begin{array}{rcl}
\mathsf{sorted}(\mathsf{x}) & \Rightarrow & \mathsf{sorted}(\mathsf{x}, \mathsf{dir}) \\
\mathsf{smaller}(\mathsf{l}, \mathsf{a}) & \Rightarrow & \mathsf{smaller}(\mathsf{a}, \mathsf{l}, \mathsf{dir}) \\
\mathsf{sortedp}(\mathsf{y.l}) & \Rightarrow & \mathsf{sortedp}(\mathsf{y.l}, \mathsf{dir}) \\
\mathsf{sortedp}(\mathsf{l}) & \Rightarrow & \mathsf{sortedp}(\mathsf{l}, \mathsf{dir})
\end{array}$$

Note that the ability of second-order variables to abstract context is particularly important to the transform

$$\mathsf{sortedp}(u) \Rightarrow \mathsf{sortedp}(u, \mathsf{dir})$$

Generalization from Case 1:

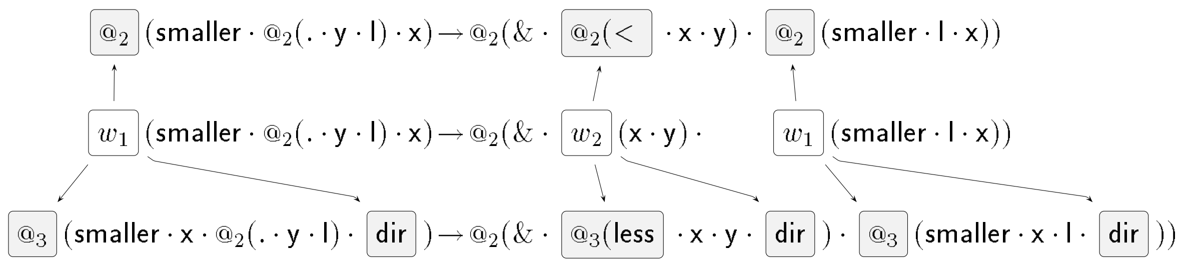


$\boxed{@_2}$ (smaller · $\boxed{@_2(.\cdot y \cdot l)}$ · x) → $\boxed{@_2(\& \cdot @_2(< \cdot x \cdot y) \cdot @_2(\text{smaller} \cdot l \cdot x))}$
$\boxed{w_1}$ (smaller · x) → $\boxed{w_2}$
$\boxed{@_3}$ (smaller · x · $\boxed{\text{Nil} \cdot \text{dir}}$) → $\boxed{\text{True}}$

Resulting transforms (indexed by the variable generating them):

$$\begin{array}{lrcl} w_1: & @_2(x_1 \cdot @_2(. \cdot \mathsf{y} \cdot \mathsf{l}) \cdot x_2) & \Rightarrow & @_3(x_1 \cdot x_2 \cdot \mathsf{Nil} \cdot \mathsf{dir}) \\ w_2: & @_2(\& \cdot @_2(< \cdot \mathsf{x} \cdot \mathsf{y}) \cdot @_2(\mathsf{smaller} \cdot \mathsf{l} \cdot \mathsf{x})) & \Rightarrow & \mathsf{True} \end{array}$$

Generalization from Case 2:



$\boxed{@_2}$ (smaller · $@_2(.\cdot y \cdot l)$ · x) → $@_2(\&$ · $\boxed{@_2(<}$ · x · y) · $\boxed{@_2}$ (smaller · l · x))
$\boxed{w_1}$ (smaller · $@_2(.\cdot y \cdot l)$ · x) → $@_2(\&$ · $\boxed{w_2}$ (x · y) · $\boxed{w_1}$ (smaller · l · x))
$\boxed{@_3}$ (smaller · x · $@_2(.\cdot y \cdot l)$ · $\boxed{\text{dir}}$) → $@_2(\&$ · $\boxed{@_3(\text{less}}$ · x · y · $\boxed{\text{dir}}$) · $\boxed{@_3}$ (smaller · x · l · $\boxed{\text{dir}}$))

Resulting transforms:

$$\begin{array}{lrcl} w_1: & @_2(x_1 \cdot x_2 \cdot x_3) & \Rightarrow & @_3(x_1 \cdot x_3 \cdot x_2 \cdot \mathsf{dir}) \\ w_2: & @_2(< \cdot x_1 \cdot x_2) & \Rightarrow & @_3(\mathsf{less} \cdot x_1 \cdot x_2 \cdot \mathsf{dir}) \end{array}$$

Figure 6.4: Detailed generalizations from matching smaller cases.

```
*Focus:* sortedp(x, dir) = sorted(x, dir)
[*closed with program*:]
sortedp(Nil, dir) → True
sortedp(a.Nil, dir) → sortedp(Nil, dir)
sortedp(a.y.l1, dir) → less(a, y, dir) & sortedp(y.l1, dir)

   *cases from* sorted(x, dir):

      *1.* case x == Nil
      sortedp(Nil, dir) = True
      sortedp(Nil, dir) = True

      *2.* case x == a.l
      sortedp(a.l, dir) = smaller(a, l, dir) & sorted(l, dir)
      sortedp(a.l, dir) = smaller(a, l, dir) & sortedp(l, dir)

         *cases from* smaller(a, l, dir):

         *1.* *case* l == Nil
            sortedp(a.Nil, dir) = True & sortedp(Nil, dir)
            sortedp(a.Nil, dir) = sortedp(Nil, dir)

         *2.* *case* l == y.l1
            sortedp(a.y.l1, dir) = less(a, y, dir) & smaller(a, l1, dir) & sortedp(y.l1, dir)
            sortedp(a.y.l1, dir) = less(a, y, dir) & sortedp(y.l1, dir)


*Prove:* smaller(a, l, dir) {less(a, y, dir) & sortedp(y.l, dir)} = True
[*closed with properties*:]
smaller(a, l, dir) {sortedp(y.l, dir) & less(a, y, dir)} = True

   *cases from* smaller(a, l, dir): ...

         *cases from* sortedp(y.l1, dir): ...

         [... *remainder of proof omitted* ...]
```

Figure 6.5: Derivation of modified version of sortedp.

so that applying it to both $\mathsf{sortedp(y.l)}$ and $\mathsf{sortedp(l)}$ gives the correct results. Secondly, transforms are used to update the property

$$\mathsf{smaller(l,a)}\ \{\mathsf{a} < \mathsf{y}\ \&\ \mathsf{sortedp(y.l)}\} = \mathsf{True}$$

to

$$\mathsf{smaller(a,l,dir)}\ \{\mathsf{less(a,y,dir)}\ \&\ \mathsf{sortedp(y.l,dir)}\} = \mathsf{True}$$

That is, they introduce the parameter dir in the appropriate places. This use of analogy is especially important since without the property, the new version of $\mathsf{sortedp}$ would be no more efficient than sorted.

Note that first-order generalization would have failed in Example 6.21, even when using the $@_n$ notation. While matching the old and new definitions of $\mathsf{smaller}$ would have worked (because of the common occurrence of & at the topmost position), the resulting transforms would be

$$\{\mathsf{smaller(y.l,x)} \Rightarrow \mathsf{smaller(x,y.l,dir)},\ \mathsf{x} < \mathsf{y}\&\mathsf{smaller(l,x)} \Rightarrow \mathsf{less(x,y,dir)}\&\mathsf{smaller(l,x,dir)}\}$$

Likewise, the transform from the definition of sorted would be

$$\mathsf{sorted(x)} \Rightarrow \mathsf{sorted(x,dir)}$$

and the transform from the focus specification would be

$$\mathsf{sortedp(x)} \Rightarrow \mathsf{sortedp(x,dir)}$$

These do match one of the `expand` terms, $\mathsf{sorted(x)}$, but fail to match any of the other terms: $\mathsf{smaller(l,a)}$, $\mathsf{sortedp(y.l)}$, and $\mathsf{sortedp(l)}$. Likewise, the transforms have no effect on the property

$$\mathsf{smaller(l,a)}\ \{\mathsf{a} < \mathsf{y}\ \&\ \mathsf{sortedp(y.l)}\} = \mathsf{True}$$

Thus the transforms from first-order generalization fail because they cannot abstract contexts.

Example 6.21 illustrated using generalization to transform terms. The next generalization uses generalization to find the closest match between sets of terms and illustrates using replay for derivation-by-analogy.

Example 6.22 The function pow raises a number to an integral power:

```
pow(x, 0)     → S(0)
pow(x, S(y)) → x * pow(x, y)
```

We will use the derivation of a program for pow to guide deriving a program for revzip, a function which joins two lists in reverse order by alternating between the lists:

$$\begin{array}{ll}
\mathsf{revzip(Nil,\ Nil)} & \rightarrow \mathsf{Nil} \\
\mathsf{revzip(a.as,\ Nil)} & \rightarrow \mathsf{revzip(as,\ Nil)} +\!\!+\ \mathsf{a.Nil} \\
\mathsf{revzip(Nil,\ b.bs)} & \rightarrow \mathsf{revzip(Nil,\ bs)} +\!\!+\ \mathsf{b.Nil} \\
\mathsf{revzip(a.as,\ b.bs)} & \rightarrow \mathsf{revzip(as,\ bs)} +\!\!+\ \mathsf{a.b.Nil}
\end{array}$$

The function ++ is an infix version of append as described in Section 3.1. Figure 6.6a shows the derivation of a program fastpow to compute pow, while Figure 6.6b shows the corresponding derivation of rzip, a program to compute revzip. The primary improvement in both cases is that the resulting program is *tail-recursive*; that is, it is in a form which allows it to be compiled into a loop [ASS85]. In the case of revzip, we also remove the calls to ++ to avoid an extra pass through the list.

The key steps in the derivation of fastpow are expanding pow, explicitly rewriting $*$ in the second case, and then choosing the right alternative. The derivation for rzip is similar: expand revzip, explicitly rewrite ++, and then choose the appropriate result. Generalizing the focus specifications,

$$\mathsf{fastpow(x, n, a)} = \mathsf{pow(x, n)} * \mathsf{a}$$

and

$$\mathsf{rzip(u, v, a)} = \mathsf{revzip(u, v)} +\!\!+\ \mathsf{a}$$

gives the transforms

$$\{\mathsf{x} \Rightarrow \mathsf{u},\ \mathsf{n} \Rightarrow \mathsf{v},\ \mathsf{fastpow} \Rightarrow \mathsf{rzip},\ * \Rightarrow +\!\!+\ ,\ \mathsf{pow} \Rightarrow \mathsf{revzip}\} \tag{6.3}$$

This is used to update the `expand` script step. Similarly, $*$ is transformed to ++ in the `rewrite` operation. The interesting part is handling the cases after expansion. Note that the new derivation has more cases, and ++ must be rewritten explicitly in cases 2–4, so matching cases is particularly important for this example.

Consider finding the appropriate match for case 3,

$$\mathsf{rzip(Nil \cdot .(b \cdot bs) \cdot a)} = +\!\!+(+\!\!+(\mathsf{revzip(Nil \cdot bs) \cdot .(b \cdot Nil)) \cdot a})$$

Applying the transforms in (6.3) to cases 1 and 2 in the fastpow derivation gives the alternatives

i. $\mathsf{rzip(u \cdot 0 \cdot a)} = +\!\!+\ (\mathsf{S\,0 \cdot a})$

ii. $\mathsf{rzip(u \cdot S\,y \cdot a)} = +\!\!+\ (+\!\!+\ (\mathsf{u} \cdot \mathsf{revzip(u \cdot y)}) \cdot \mathsf{a})$

```
Focus: fastpow(x, n, a) =
          pow(x, n) * a
[closed with program:]
fastpow(x, 0, a) → a
fastpow(x, S(y), a) →
          fastpow(x, y, x * a)

  cases from pow(x, n):

  1. case n == 0
     fastpow(x, 0, a) = S(0) * a
     fastpow(x, 0, a) = a

  2. case n == S(y)
     fastpow(x, S(y), a) =
         x * pow(x, y) * a
     fastpow(x, S(y), a) =
         fastpow(x, y, x * a)
```

a. The derivation of fastpow.

```
Focus: rzip(u, v, a) = revzip(u, v) ++ a
[closed with program:]
rzip(Nil, Nil, a) → a
rzip(a1.as, Nil, a) → rzip(as, Nil, a1.a)
rzip(Nil, b.bs, a) → rzip(Nil, bs, b.a)
rzip(a1.as, b.bs, a) → rzip(as, bs, a1.b.a)

  cases from revzip(u, v):

  1. case v == Nil & u == Nil
     rzip(Nil, Nil, a) = Nil ++ a
     rzip(Nil, Nil, a) = a

  2. case v == Nil & u == a1.as
     rzip(a1.as, Nil, a) =
         revzip(as, Nil) ++ a1.Nil ++ a
     rzip(a1.as, Nil, a) = rzip(as, Nil, a1.a)

  3. case v == b.bs & u == Nil
     rzip(Nil, b.bs, a) =
         revzip(Nil, bs) ++ b.Nil ++ a
     rzip(Nil, b.bs, a) = rzip(Nil, bs, b.a)

  4. case v == b.bs & u == a1.as
     rzip(a1.as, b.bs, a) =
         revzip(as, bs) ++ a1.b.Nil ++ a
     rzip(a1.as, b.bs, a) = rzip(as, bs, a1.b.a)
```

b. The derivation of rzip.

Figure 6.6: Using replay for derivation-by-analogy.

Applying Algorithm 6.20 results in the terms and transforms

i. $\mathsf{rzip}(w_1 \cdot w_2 \cdot \mathsf{a}) = \mathbin{+\!\!+} (w_3 \cdot \mathsf{a})$

$\{\mathsf{Nil} \Rightarrow \mathsf{u},\ .(\mathsf{b} \cdot \mathsf{bs}) \Rightarrow 0, \mathbin{+\!\!+} (\mathsf{revzip}(\mathsf{Nil} \cdot \mathsf{bs}) \cdot .(\mathsf{b} \cdot \mathsf{Nil})) \Rightarrow \mathsf{S}\,0\}$

ii. $\mathsf{rzip}(w_1 \cdot w_2 \cdot \mathsf{a}) = \mathbin{+\!\!+} (w_3(\mathbin{+\!\!+} \cdot \mathsf{revzip}(w_1 \cdot w_4)) \cdot \mathsf{a})$

$\{\mathsf{Nil} \Rightarrow \mathsf{u},\ .(\mathsf{b} \cdot \mathsf{bs}) \Rightarrow \mathsf{S}\,\mathsf{y}, x_1(x_2 \cdot .(\mathsf{b} \cdot \mathsf{Nil})) \Rightarrow x_1(\mathsf{u} \cdot x_2),\ \mathsf{bs} \Rightarrow \mathsf{y}\}$

For the detailed versions of the generalizations, see Figure 6.7. The norm of (i) is $18/(12+22) \approx 53\%$, while the norm of (ii) is $18/(18+22) = 45\%$. This leads ReFocus to apply the script from the second case of Figure 6.6a, as desired. Second-order generalization worked because it noted that revzip occurred in both equations, that ++ occurred twice in both equations, and that Nil was matched to u twice. The other cases are similar, except that the first case of Figure 6.6a is used for the first case of Figure 6.6b.

The next step for case 3 is to transform the explicit rewrite in case 2 of Figure 6.6a to `rewrite(++)` and then execute the step to get the alternatives

i. $\mathsf{rzip}(\mathsf{Nil} \cdot .(\mathsf{b} \cdot \mathsf{bs}) \cdot \mathsf{a}) = \mathsf{rzip}(\mathsf{Nil} \cdot \mathsf{bs} \cdot .(\mathsf{b} \cdot \mathsf{a}))$

ii. $\mathsf{rzip}(\mathsf{Nil} \cdot .(\mathsf{b} \cdot \mathsf{bs}) \cdot \mathsf{a}) = \mathbin{+\!\!+}(\mathsf{rzip}(\mathsf{Nil} \cdot \mathsf{bs} \cdot .(\mathsf{b} \cdot \mathsf{Nil})) \cdot \mathsf{a})$

Applying the transforms from the focus specification to the original result,

$$\mathsf{fastpow}(\mathsf{x} \cdot \mathsf{S}\,\mathsf{y} \cdot \mathsf{a}) = \mathsf{fastpow}(\mathsf{x} \cdot \mathsf{y} \cdot *(\mathsf{x} \cdot \mathsf{a}))$$

gives[8]

$$\mathsf{rzip}(\mathsf{u} \cdot \mathsf{Nil} \cdot \mathsf{a}) = \mathsf{rzip}(\mathsf{u} \cdot \mathsf{y} \cdot \mathbin{+\!\!+}(\mathsf{u} \cdot \mathsf{a}))$$

Algorithm 6.20 gives

i. $\mathsf{rzip}(w_1 \cdot w_2 \cdot \mathsf{a}) = \mathsf{rzip}(w_1 \cdot w_4 \cdot w_5(w_6 \cdot \mathsf{a}))$

$\{\mathsf{u} \Rightarrow \mathsf{Nil},\ \mathsf{Nil} \Rightarrow .(\mathsf{b} \cdot \mathsf{bs}),\ \mathsf{y} \Rightarrow \mathsf{bs}, \mathbin{+\!\!+} \Rightarrow .,\ \mathsf{u} \Rightarrow \mathsf{b}\}$

ii. $\mathsf{rzip}(w_1 \cdot w_2 \cdot \mathsf{a}) = w_3(\mathsf{rzip}(w_1 \cdot w_4 \cdot w_5(w_6 \cdot w_7)))$

$\{\mathsf{u} \Rightarrow \mathsf{Nil},\ \mathsf{Nil} \Rightarrow .(\mathsf{b} \cdot \mathsf{bs}),\ x_1 \Rightarrow \mathbin{+\!\!+} (x_1 \cdot \mathsf{a}),\ \mathsf{y} \Rightarrow \mathsf{bs}, \mathbin{+\!\!+} \Rightarrow .,\ \mathsf{u} \Rightarrow \mathsf{b},\ \mathsf{a} \Rightarrow \mathsf{Nil}\}$

For the detailed versions of the generalizations, see Figure 6.8. The norm of (i) is $13/31 \approx 42\%$, while the norm of (ii) is $18/34 \approx 53\%$, leading ReFocus to choose (i) as desired. The primary difference between the two sets of transforms is that $x_1 \Rightarrow \mathbin{+\!\!+} (x_1 \cdot \mathsf{a})$ appeared in the second

[8]The rule $\mathtt{S\,y} \Rightarrow \mathtt{Nil}$ comes from comparing the initial states of the old and new cases after the expand step.

Generalizing the left-hand side of Case 3 against Case 1:

$$\begin{tikzcd}
@_3(\mathsf{rzip} \cdot \boxed{\mathsf{Nil}} \cdot \boxed{@_2(. \cdot \mathsf{b} \cdot \mathsf{bs})} \cdot \mathsf{a}) \\
@_3(\mathsf{rzip} \cdot \boxed{w_1} \cdot \boxed{w_2} \cdot \mathsf{a}) \arrow[u, shift left] \arrow[u, shift right] \arrow[d, shift left] \arrow[d, shift right] \\
@_3(\mathsf{rzip} \cdot \boxed{\mathsf{u}} \cdot \boxed{\mathsf{0}} \cdot \mathsf{a})
\end{tikzcd}$$

Generalizing the right-hand side of Case 3 against Case 1:

$$\begin{tikzcd}
@_2(+\!\!+ \cdot \boxed{@_2(+\!\!+ \cdot @_2(\mathsf{revzip} \cdot \mathsf{Nil} \cdot \mathsf{bs}) \cdot @_2(. \cdot \mathsf{b} \cdot \mathsf{Nil}))} \cdot \mathsf{a}) \\
@_2(+\!\!+ \cdot \boxed{w_3} \cdot \mathsf{a}) \arrow[u] \arrow[d] \\
@_2(+\!\!+ \cdot \boxed{@_1(\mathsf{S} \cdot \mathsf{0})} \cdot \mathsf{a})
\end{tikzcd}$$

Resulting transforms:

$$\begin{array}{lrcl}
w_1 : & \mathsf{Nil} & \Rightarrow & \mathsf{u} \\
w_2 : & @_2(. \cdot \mathsf{b} \cdot \mathsf{bs}) & \Rightarrow & \mathsf{0} \\
w_3 : & @_2(+\!\!+ \cdot @_2(\mathsf{revzip} \cdot \mathsf{Nil} \cdot \mathsf{bs}) \cdot @_2(. \cdot \mathsf{b} \cdot \mathsf{Nil})) & \Rightarrow & @_1(\mathsf{S} \cdot \mathsf{0})
\end{array}$$

Generalizing the left-hand side of Case 3 against Case 2: nearly identical to Case 1.

Generalizing the right-hand side of Case 3 against Case 2:

$$\begin{tikzcd}
@_2(+\!\!+ \cdot \boxed{@_2}(+\!\!+ \cdot @_2(\mathsf{revzip} \cdot \boxed{\mathsf{Nil}} \cdot \boxed{\mathsf{bs}}) \cdot \boxed{@_2(. \cdot \mathsf{b} \cdot \mathsf{Nil})}) \cdot \mathsf{a}) \\
@_2(+\!\!+ \cdot \boxed{w_3}(+\!\!+ \cdot @_2(\mathsf{revzip} \cdot \boxed{w_1} \cdot \boxed{w_4})) \cdot \mathsf{a}) \arrow[u, shift left=3] \arrow[u, shift left] \arrow[u, shift right] \arrow[u, shift right=3] \arrow[d, shift left=3] \arrow[d, shift left] \arrow[d, shift right] \arrow[d, shift right=3] \\
@_2(+\!\!+ \cdot \boxed{@_2}(+\!\!+ \cdot \boxed{\mathsf{u}} \cdot @_2(\mathsf{revzip} \cdot \boxed{\mathsf{u}} \cdot \boxed{\mathsf{y}})) \cdot \mathsf{a})
\end{tikzcd}$$

Resulting transforms:

$$\begin{array}{lrcl}
w_1 : & \mathsf{Nil} & \Rightarrow & \mathsf{u} \\
w_2 : & @_2(. \cdot \mathsf{b} \cdot \mathsf{bs}) & \Rightarrow & @_1(\mathsf{S} \cdot \mathsf{y}) \\
w_3 : & @_2(x_1 \cdot x_2 \cdot @_2(. \cdot \mathsf{b} \cdot \mathsf{Nil})) & \Rightarrow & @_2(x_1 \cdot \mathsf{u} \cdot x_2) \\
w_4 : & \mathsf{bs} & \Rightarrow & \mathsf{y}
\end{array}$$

Figure 6.7: Detailed generalizations from matching case 3 against cases 1 and 2 of fastpow.

Generalizing with case (i):

$$\begin{tikzcd}
@_3(\mathsf{rzip}\,\cdot & \boxed{\mathsf{u}} & \cdot & \boxed{\mathsf{Nil}} & \cdot\,\mathsf{a}) \;=\; @_3(\mathsf{rzip}\,\cdot & \boxed{\mathsf{u}} & \cdot & \boxed{\mathsf{y}} & \cdot\, @_2(& \boxed{+\!\!+} & \cdot & \boxed{\mathsf{u}} & \cdot\,\mathsf{a})) \\
@_3(\mathsf{rzip}\,\cdot & \boxed{w_1} \arrow[u] \arrow[d] & \cdot & \boxed{w_2} \arrow[u] \arrow[d] & \cdot\,\mathsf{a}) \;=\; @_3(\mathsf{rzip}\,\cdot & \boxed{w_1} \arrow[u] \arrow[d] & \cdot & \boxed{w_4} \arrow[u] \arrow[d] & \cdot\, @_2(& \boxed{w_5} \arrow[u] \arrow[d] & \cdot & \boxed{w_6} \arrow[u] \arrow[d] & \cdot\,\mathsf{a})) \\
@_3(\mathsf{rzip}\,\cdot & \boxed{\mathsf{Nil}} & \cdot & \boxed{@_2(.\cdot\mathsf{b}\cdot\mathsf{bs})} & \cdot\,\mathsf{a}) \;=\; @_3(\mathsf{rzip}\,\cdot & \boxed{\mathsf{Nil}} & \cdot & \boxed{\mathsf{bs}} & \cdot\, @_2(& \boxed{.} & \cdot & \boxed{\mathsf{b}} & \cdot\,\mathsf{a}))
\end{tikzcd}$$

Resulting transforms:

$$\{\mathsf{u} \Rightarrow \mathsf{Nil},\ \mathsf{Nil} \Rightarrow @_2(.\cdot\mathsf{b}\cdot\mathsf{bs}),\ \mathsf{y} \Rightarrow \mathsf{bs}, +\!\!+\ \Rightarrow .,\ \mathsf{u} \Rightarrow \mathsf{b}\}$$

Generalizing with case (ii):

$$\begin{tikzcd}
@_3(\mathsf{rzip}\,\cdot & \boxed{\mathsf{u}} & \cdot & \boxed{\mathsf{Nil}} & \cdot\,\mathsf{a}) = @_3(\mathsf{rzip}\,\cdot & & & \boxed{\mathsf{u}} & \cdot & \boxed{\mathsf{y}} & \cdot\, @_2(& \boxed{+\!\!+} & \cdot & \boxed{\mathsf{u}} & \cdot & \boxed{\mathsf{a}} &)) & & \\
@_3(\mathsf{rzip}\,\cdot & \boxed{w_1} \arrow[u] \arrow[d] & \cdot & \boxed{w_2} \arrow[u] \arrow[d] & \cdot\,\mathsf{a}) = & \boxed{w_3} \arrow[d] \arrow[drrrrrrrrrrrr] & (@_3(\mathsf{rzip}\,\cdot & \boxed{w_1} \arrow[u] \arrow[d] & \cdot & \boxed{w_4} \arrow[u] \arrow[d] & \cdot\, @_2(& \boxed{w_5} \arrow[u] \arrow[d] & \cdot & \boxed{w_6} \arrow[u] \arrow[d] & \cdot & \boxed{w_7} \arrow[u] \arrow[d] &))) & & \\
@_3(\mathsf{rzip}\,\cdot & \boxed{\mathsf{Nil}} & \cdot & \boxed{@_2(.\cdot\mathsf{b}\cdot\mathsf{bs})} & \cdot\,\mathsf{a}) = & \boxed{@_2(+\!\!+} & \cdot\, @_3(\mathsf{rzip}\,\cdot & \boxed{\mathsf{Nil}} & \cdot & \boxed{\mathsf{bs}} & \cdot\, @_2(& \boxed{.} & \cdot & \boxed{\mathsf{b}} & \cdot & \boxed{\mathsf{Nil}} &))\,\cdot & \boxed{\mathsf{a}} &)
\end{tikzcd}$$

Resulting transforms:

$$\{\mathsf{u} \Rightarrow \mathsf{Nil},\ \mathsf{Nil} \Rightarrow @_2(.\cdot\mathsf{b}\cdot\mathsf{bs}),\ x_1 \Rightarrow @_2(+\!\!+\ \cdot x_1\cdot\mathsf{a}),\ \mathsf{y} \Rightarrow \mathsf{bs}, +\!\!+\ \Rightarrow .,\ \mathsf{u} \Rightarrow \mathsf{b},\ \mathsf{a} \Rightarrow \mathsf{Nil}\}$$

Figure 6.8: Detailed generalizations from matching rewrite results for rzip.

one, so second-order generalization was successful because rzip was at the head of the term in (i) and embedded in a context in (ii). Again, the other cases are similar.

Finally, Refocus must test the new derivation for acceptability. Consider case 3. Transforming the final state of the original derivation gives

$$\mathsf{rzip(u,\ Nil,\ a) = rzip(u,\ y,\ u \mathbin{+\!\!+}\ a)}$$

This is close to the actual final result,

$$\mathsf{rzip(Nil,\ b.y,\ a) = rzip(Nil,\ y,\ b.a)}$$

in that it is tail-recursive and reflects the change from numbers to lists. However, it is not an exact match, illustrating that transformations are not very effective at predicting the final form of the program. Furthermore, since the same case is used to guide the derivation of three different cases of rzip, and since each case involves significant processing, it is not clear that any given set of transforms could generate exact matches for all three cases. This issue motivates the acceptance tests described in Chapter 7.

While second-order generalization is very flexible, it does have limitations, as shown by the following example:

Example 6.23 The function rev reverses a list:

$$\mathsf{rev(Nil) \rightarrow Nil}$$
$$\mathsf{rev(a.x) \rightarrow rev(x) \mathbin{+\!\!+} a.Nil}$$

where $\mathbin{+\!\!+}$ again appends two lists. Focusing on the equation

$$\mathsf{aprev(l, a) = rev(l) \mathbin{+\!\!+} a}$$

expanding rev, rewriting $\mathbin{+\!\!+}$ in the second case, and selecting the desired result gives the derivation of a tail-recursive program for reversing a list. This is shown in Figure 6.9. The structure of this derivation is the same as that for fastpow in Figure 6.6a. Intuitively, we should be able to use the derivation of aprev to guide the derivation of fastpow. However, ReFocus fails on this example. After rewriting $*$ in the new derivation, we obtain the alternatives

i. $\mathsf{fastpow(x \cdot S(y) \cdot a) = *(fastpow(x \cdot y \cdot a) \cdot x)}$

ii. $\mathsf{fastpow(x \cdot S(y) \cdot a) = *(fastpow(x \cdot y \cdot x) \cdot a)}$

iii. $\mathsf{fastpow(x \cdot S(y) \cdot a) = fastpow(x \cdot y \cdot *(x \cdot a))}$

Focus: aprev(l, a) = rev(l) ++ a
[*closed with program*:]
aprev(Nil, a) $\rightarrow$ a
aprev(a1.x, a) $\rightarrow$ aprev(x, a1.a)

cases from rev(l):

1. *case* l == Nil
   aprev(Nil, a) = Nil ++ a
   aprev(Nil, a) = a

2. *case* l == a1.x
   aprev(a1.x, a) = rev(x) ++ a1.Nil ++ a
   aprev(a1.x, a) = aprev(x, a1.a)

Figure 6.9: Derivation of aprev.

Applying the transforms from comparing the focus specifications in Figures 6.6a and 6.9,

$$\{\mathsf{aprev}(\mathsf{l}, x_1) \Rightarrow \mathsf{fastpow}(\mathsf{x}, \mathsf{n}, x_1), \mathsf{++} \Rightarrow *, \mathsf{rev}(\mathsf{l}) \Rightarrow \mathsf{pow}(\mathsf{x}, \mathsf{n})\}$$

to

$$\mathsf{aprev}(\mathsf{a1.x}, \mathsf{a}) = \mathsf{aprev}(\mathsf{x}, \mathsf{a1.a})$$

has no effect. Thus Refocus attempts to match the new results (cases i–iii) against

$$\mathsf{aprev}(.(\mathsf{a1} \cdot \mathsf{x}) \cdot \mathsf{a}) = \mathsf{aprev}(\mathsf{x} \cdot .(\mathsf{a1} \cdot \mathsf{a}))$$

The corresponding generalization terms and transforms are

i. $w_1(\mathsf{x} \cdot \mathsf{a}) = w_2(w_3(\mathsf{x}) \cdot w_4)$

   $\{\mathsf{aprev}(.(\mathsf{a1} \cdot x_1) \cdot x_2) \Rightarrow \mathsf{fastpow}(x_1 \cdot \mathsf{S}(\mathsf{y}) \cdot x_2), \mathsf{aprev} \Rightarrow *,$
   $x_1 \Rightarrow \mathsf{fastpow}(x_1 \cdot \mathsf{y} \cdot \mathsf{a}), .(\mathsf{a1} \cdot \mathsf{a}) \Rightarrow \mathsf{i}\}$

ii. $w_1(\mathsf{x} \cdot \mathsf{a}) = w_2(w_3(\mathsf{x}) \cdot w_4(\mathsf{a}))$

   $\{\mathsf{aprev}(.(\mathsf{a1} \cdot x_1) \cdot x_2) \Rightarrow \mathsf{fastpow}(x_1 \cdot \mathsf{S}(\mathsf{y}) \cdot x_2), \mathsf{aprev} \Rightarrow *,$
   $x_1 \Rightarrow \mathsf{fastpow}(x_1 \cdot \mathsf{y} \cdot \mathsf{x}), .(\mathsf{a1} \cdot x_1) \Rightarrow x_1\}$

iii. $w_1(\mathsf{x} \cdot \mathsf{a}) = w_2(\mathsf{x} \cdot w_3(w_4 \cdot \mathsf{a}))$

   $\{\mathsf{aprev}(.(\mathsf{a1} \cdot x_1) \cdot x_2) \Rightarrow \mathsf{fastpow}(x_1 \cdot \mathsf{S}(\mathsf{y}) \cdot x_2),$
   $\mathsf{aprev}(x_1 \cdot x_2) \Rightarrow \mathsf{fastpow}(x_1 \cdot \mathsf{y} \cdot x_2), . \Rightarrow *, \mathsf{a1} \Rightarrow \mathsf{x}\}$

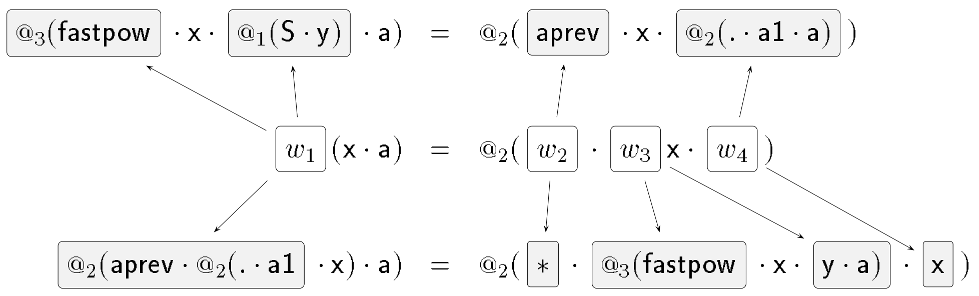
For the detailed versions of the generalizations, see Figure 6.10. ReFocus should select case (iii) since it is the tail-recursive version. Case (i) is eliminated because its norm is 64% while the norm of case (ii) is $21/37 \approx 58\%$. However, the norm of (iii) is also 58%, so generalization does not distinguish between matching against cases (ii) and (iii). The problem is that the transforms obtained from comparing the focus specifications are too specific, so applying them to the original result has no effect. In all three cases, only x, a, and $@_2$ appear in the generalization term, with a second occurrence of a differentiating between case (i) and cases (ii) and (iii). This means the transforms contain nearly all the symbols from the instance terms. In Example 6.22, ReFocus recognized the tail-recursive form by finding a trivial context for the recursive call in the selected case. But in this example, aprev was not transformed to fastpow, so ReFocus could not form the appropriate contexts and select the right result. For the same reasons, first-order generalization also fails.

However, the failure in Example 6.23 should not be surprising. Because pow has two arguments while rev has only one, the terms in the two derivations contain many structural differences. Furthermore, the two domains are very different, so there are no shared function names. Only two symbols were shared, $@_2$ and a, and these were almost incidental. When the terms in two derivations do not share a common structure and use distinct sets of symbols, there is no information for syntactic analogy to transfer. Semantic information is needed for such examples.

6.4 Conclusion

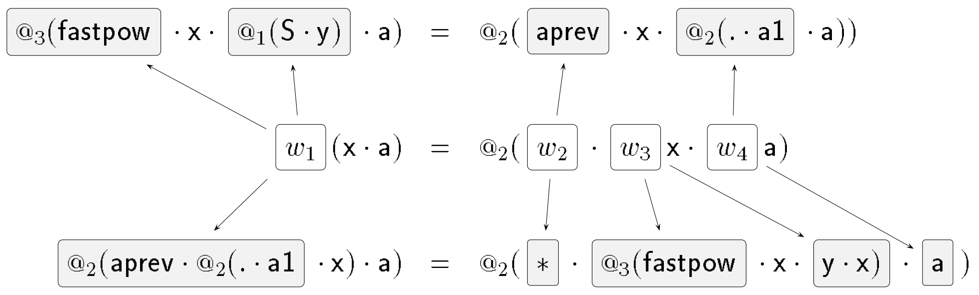
This chapter addressed various problems with applying second-order generalization to replay. The key issues were restricting the sets of generalizations and computing generalizations efficiently. These are addressed by choosing generalizations with maximally large source terms and using dynamic programming to compute such generalizations efficiently. However, computing generalizations efficiently is not enough; the computed generalizations must be useful. This was addressed by heuristics to restrict the candidate generalizations further, improve generalization across distinct domains, and add contextual information. The refined definition of generalization was shown to be useful by applying it to simple replay problems.

The examples show how generalization is used by ReFocus to transfer information between derivations. Unless the user changes both the symbols and the structure of terms, second-order generalization can be very successful in selecting between alternatives and transforming small terms. However, transforming large terms is less likely to produce correct results, as shown by the acceptance test at the end of Example 6.22. In general, second-order generalization is best at selecting the closest match between alternatives. Transformations of large terms often produce incorrect results because of the large number of details involved. However, transforming large i.


@3(fastpow · x · @1(S · y) · a) = @2(aprev · x · @2(. · a1 · a))
w1 (x · a) = @2(w2 · w3 x · w4)
@2(aprev · @2(. · a1 · x) · a) = @2(∗ · @3(fastpow · x · y · a) · x)

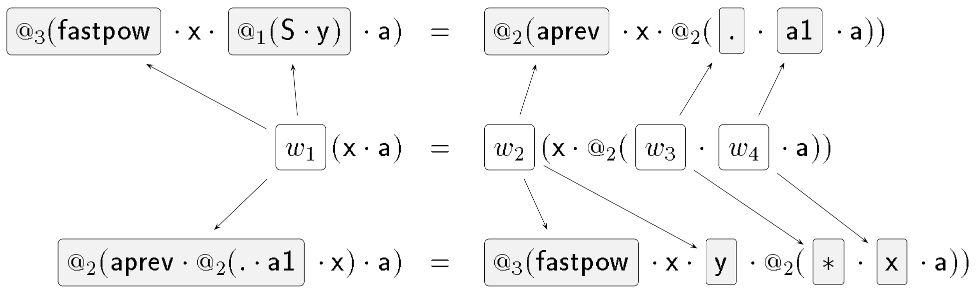
$\{@_2(\mathsf{aprev} \cdot @_2(. \cdot \mathsf{a1} \cdot x_1) \cdot x_2) \Rightarrow @_3(\mathsf{fastpow} \cdot x_1 \cdot @_1(\mathsf{S} \cdot \mathsf{y}) \cdot x_2),$
$\mathsf{aprev} \Rightarrow *,\ x_1 \Rightarrow @_3(\mathsf{fastpow} \cdot x_1 \cdot \mathsf{y} \cdot \mathsf{a}), @_2(. \cdot \mathsf{a1} \cdot \mathsf{a}) \Rightarrow \mathsf{i}\}$

ii.


@3(fastpow · x · @1(S · y) · a) = @2(aprev · x · @2(. · a1 · a))
w1 (x · a) = @2(w2 · w3 x · w4 a)
@2(aprev · @2(. · a1 · x) · a) = @2(∗ · @3(fastpow · x · y · x) · a)

$\{@_2(\mathsf{aprev} \cdot @_2(. \cdot \mathsf{a1} \cdot x_1) \cdot x_2) \Rightarrow @_3(\mathsf{fastpow} \cdot x_1 \cdot @_1(\mathsf{S} \cdot \mathsf{y}) \cdot x_2),$
$\mathsf{aprev} \Rightarrow *,\ x_1 \Rightarrow @_3(\mathsf{fastpow} \cdot x_1 \cdot \mathsf{y} \cdot \mathsf{i}), @_2(. \cdot \mathsf{a1} \cdot x_1) \Rightarrow x_1\}$

iii.


@3(fastpow · x · @1(S · y) · a) = @2(aprev · x · @2(. · a1 · a))
w1 (x · a) = w2 (x · @2(w3 · w4 · a))
@2(aprev · @2(. · a1 · x) · a) = @3(fastpow · x · y · @2(∗ · x · a))

$\{@_2(\mathsf{aprev} \cdot @_2(. \cdot \mathsf{a1} \cdot x_1) \cdot x_2) \Rightarrow @_3(\mathsf{fastpow} \cdot x_1 \cdot @_1(\mathsf{S} \cdot \mathsf{y}) \cdot x_2),$
$@_2(\mathsf{aprev} \cdot x_1 \cdot x_2) \Rightarrow @_3(\mathsf{fastpow} \cdot x_1 \cdot \mathsf{y} \cdot x_2),\ . \Rightarrow *,\ \mathsf{a1} \Rightarrow \mathsf{x}\}$

Figure 6.10: Detailed generalizations from matching rewrite results for fastpow.

terms is usually useful only when comparing the results of derivations. Chapter 7 presents an alternative, and more robust, method for comparing results.

Chapter 7

Acceptance Testing

Chapters 4, 5, and 6 describe transforming derivation sequences to apply them to new specifications. This allows them to be replayed with minimal intervention from the user. For theorem proving, reexecuting derivations is sufficient. But for synthesizing programs, replay must do more. A program derivation is developed for a purpose, and replaying it should fulfill the same purpose. If it does not, the user needs to know so that the derivation can be repaired. But expecting the user to recheck the results manually is not reasonable. Maintenance is a constant activity which starts as soon as the initial specification is written, so derivations are replayed often. Thus replay must automatically test the results to verify that they are acceptable.

The obvious method is to compare the old and new programs and warn the user if they are different. But a simple comparison is too strict; small differences, such as renaming variables and reordering arguments, would lead to warnings. Using analogical reasoning to propagate differences might improve the test, but examples such as 6.22 show that applying transforms is unlikely to produce an exact match. A more lenient test is needed. However, there must be a balance. If the test is too lenient, then the user cannot rely on replay to catch errors. If the test is too conservative, then replay will produce too many false warnings. In either case, the user spends too much time reexamining results.

There are other shortcomings with requiring an exact match between derived programs. One problem is that rewrite steps often make only small gains in efficiency, and failing to repeat such steps should be ignored. For example, rewriting '$\mathsf{S}(0) * \mathsf{a}$' to '$\mathsf{a}$' in case 1 of the fastpow derivation (Example 6.22) gives only a constant-time improvement. Failing to repeat such a step is not significant. Another problem is that comparing results provides little assistance in repairing derivations. It can say that a result is not acceptable, but it cannot indicate what is wrong. However, the fundamental problem with requiring an exact match is that it ignores the goals of constructing derivations. Abstractly, replay fails exactly when the new derivation does not fulfill the goals of the original derivation. An ideal test would identify the goals of the original derivation and verify that the new derivation satisfies those goals. Unfortunately, capturing information about goals is difficult.

This chapter describes a heuristic for capturing information about goals based on *term orderings*. These are well-founded orderings used to orient rules in rewrite systems, such as Focus, so that applying a rule always decreases the size of a term. Such orderings are used to show that rewriting terminates [DJ90] and that inductive proofs based on rewriting are sound [Red90b, BRH94]. But because term orderings indicate when a program derivation is finished, they can also be used to capture information about goals.

The difference between a specification and a program is that a program is orientable while a specification is not. Initially, a function is defined by an unoriented equation. A derivation turns this equation into a set of oriented rules. This results in an executable program. Thus the steps in a derivation make progress by removing subterms which block an equation from being oriented. These subterms are called *precedence violations*. This chapter shows that precedence violations can be used to measure progress during replay and provide a basis for capturing information about goals. This allows ReFocus to check progress without requiring the user to supply extra information.

Section 7.1 presents the term ordering used in Focus. This ordering is used to determine which parts of a specification block orientation. Section 7.2 presents an algorithm for computing precedence violations. Finally, Section 7.3 describes using violations in replay and gives examples. While this chapter describes using precedence violations in ReFocus, these techniques can be applied to any transformational implementation system based on oriented rewriting.

7.1 Ordering Terms

There are many methods for ordering terms in rewrite systems; see [Der87, DJ90] for surveys. This section describes the method used in Focus, the *lexicographic recursive path ordering* $\succ$ [Der87, KL80].[1] It combines a relation over functions, multiset orderings, lexicographic orderings, and the subterm relationship.

We use the following notation in this chapter. The set $\{i, i+1, \ldots, j\}$ is denoted by $i..j$. The sequence $\langle x_1, \ldots, x_n \rangle$ is denoted by $\overline{x_n}$. The symbol $>$ denotes a partial order, that is, a transitive, irreflexive relation. We say that a partial order $>$ is *well-founded* if there is no infinite descending sequence $x_1 > x_2 > \ldots$. The symbol $\cong$ denotes an equivalence relation. The symbol $\geq$ denotes the preorder defined by $x \geq y \iff x > y$ or $x \cong y$. Given a partial order $>$, $\gg$ denotes its multiset[2] extension and $>^*$ its lexicographic extension. If $\uplus$ denotes

[1] This ordering is also known, in a slightly generalized form, as the *recursive path ordering with status*; *cf.* [Ste89].

[2] A *multiset* is a set in which an element can appear multiple times.

multiset union, then $\gg$ is defined by

$$X \uplus \{x\} \gg Y \uplus \{y_1, \ldots, y_k\} \iff x > y_i \text{ for all } i \text{ and either } X = Y \text{ or } X \gg Y$$

The lexicographic extension, $>^*$, is defined by

$$\langle u_1, \ldots, u_n\rangle >^* \langle v_1, \ldots, v_m\rangle$$

if for some k in $1..n$,

$$u_1 \cong v_1, \ldots, u_{k-1} \cong v_{k-1}, \text{ and } u_k > v_k$$

For terms, we use the symbols $\succ$, $\approx$, $\succeq$, $\succ\!\!\succ$, and $\succ^*$.

Definition 7.1 (Lexicographic Recursive Path Ordering) Let

- $>$ be a well-founded partial ordering on function symbols,
- $\cong$ be an equivalence relation on function symbols, and
- $f(u_1, \ldots, u_n) \approx g(v_1, \ldots, v_n)$ be the equivalence relation over terms defined by $f \cong g$ and $u_i \approx v_i$ for each i.

Then $f(u_1, \ldots, u_n) = u \succ v = g(v_1, \ldots, v_m)$ if and only if at least one of the following is true:

- $u_i \succeq v$ for some i in $1..n$,
- $f > g$ and $u \succ v_i$ for all i in $1..m$,
- $f \cong g$, both are lexicographically ordered, $n = m$, $\langle u_1, \ldots, u_n\rangle \succ^* \langle v_1, \ldots, v_n\rangle$, and $u \succ v_i$ for all i in $1..n$, or
- $f \cong g$, neither is lexicographically ordered, and $\{u_1, \ldots, u_n\} \succ\!\!\succ \{v_1, \ldots, v_m\}$.

In Focus, the user specifies $>$ and $\cong$ by explicit precedence declarations. All builtin functions such as if are assumed to be smaller than functions defined by the user. In turn, all constructor symbols such as S and Nil are assumed to be smaller than all functions. The user also specifies which functions are ordered lexicographically and the order in which the arguments are to be compared;[3] by default, there is no lexicographic ordering on a function.

Examples 7.2

[3] For simplicity, however, we assume in this chapter that if a function is lexicographically ordered, it is ordered from left to right.

1. If $* > \mathsf{fastpow}$, then

$$\mathsf{pow(x,n)} * \mathsf{a} \succ \mathsf{fastpow(x,n,a)}$$

2. If instead $\mathsf{fastpow} > *$ and if $\mathsf{fastpow}$ is lexicographically ordered from left to right, then

$$\mathsf{fastpow(x,S(y),a)} \succ \mathsf{fastpow(x,y,x} * \mathsf{a)}$$

because $\mathsf{S(y)} \succ \mathsf{y}$.

3. If $\mathsf{fastpow} \cong \mathsf{myfun}$ and neither is ordered lexicographically, then

$$\mathsf{fastpow(S(c),S(b),a)} \succ \mathsf{myfun(b,c,b,a)}$$

by $\ggcurly$.

Definition 7.1 does not restrict $>$ beyond requiring that it be a well-founded partial ordering. But program derivation places other constraints on $>$. Since the goal of program derivation is to improve efficiency, $>$ should be specified so that if $s \succ t$, rewriting s to t improves efficiency. However, there is no absolute notion of efficiency. Usually, the user is concerned with reducing time and space, but the combination that is appropriate to a particular problem and how to achieve it must be decided by the user. This leads to viewing $>$ as a *preference relation*. That is, if $\mathsf{g} > \mathsf{f}$, then the user has declared that terms involving f are to be preferred over those containing g. The operational meaning is that Focus should replace g by f whenever it can do so by rewriting, assuming that f is preferred because it improves efficiency in some way. Thus we can also view $>$ as a preference relation.

Consider Example 7.21. Making $\mathsf{fastpow}$ smaller than pow and $*$ allows orienting the inductive hypothesis for $\mathsf{fastpow}$ as

$$\mathsf{pow(x,n)} * \mathsf{a} \rightarrow \mathsf{fastpow(x,n,a)}$$

so that occurrences of pow and $*$ are rewritten to $\mathsf{fastpow}$. This is the key to introducing tail-recursion in the program for $\mathsf{fastpow}$. Furthermore, if we prove that

$$\mathsf{pow(x,n)} = \mathsf{fastpow(x,n,0)}$$

then $\mathsf{pow} > \mathsf{fastpow}$ leads to orienting the equation as

$$\mathsf{pow(x,n)} \rightarrow \mathsf{fastpow(x,n,0)}$$

Thus preferring $\mathsf{fastpow}$ to pow leads to a rewrite rule which improves the efficiency of programs by replacing calls to pow by calls to $\mathsf{fastpow}$.

Viewing $>$ as a preference relation means the term ordering forms a useful basis for capturing information about goals. For example, if $\mathsf{f} > \mathsf{g} > \mathsf{h}$ and we have the equation

$$\mathsf{g}(\overline{u}) = \mathsf{f}(\overline{v}) \tag{7.1}$$

then applying the rewrite step

$$\mathsf{f}(\overline{v}) \to \mathsf{h}(\overline{v}) \tag{7.2}$$

makes progress by replacing f by h to obtain an equation, $\mathsf{g}(\overline{u}) = \mathsf{h}(\overline{v})$, which is orientable at the topmost position. More abstractly, we prefer expressions containing h over expressions containing f, so applying (7.2) brings us closer to the goal of finding the preferred implementation for (7.1). Thus we say that the goal of applying $\mathsf{f}(\overline{v}) \to \mathsf{h}(\overline{v})$ was to remove the precedence violation $\mathsf{g} \not> \mathsf{f}$. In this way, $>$ captures information about goals. We formalize this in the next section.

7.2 Precedence Violations

Informally, a pair of terms is a precedence violation if it blocks an equation from being oriented. This section formalizes this by giving a set of functions which compute violations for a pair of terms s and t. These functions loosely parallel $\succ$. Each condition for $\succ$ depends on the relationship between the topmost symbols; if the topmost symbols of s and t match the relationship but do not satisfy the corresponding conditions, then the unsatisfied conditions are used to generate information about violations.

There is no absolute definition of precedence violations since it is not always clear why an equation cannot be oriented. For example, suppose $\mathsf{f} > \mathsf{g} \cong \mathsf{g}' > \mathsf{h}$ where g and g' are lexicographically oriented from left to right, and consider the equation

$$\mathsf{g}(\mathsf{S}(\mathsf{a}), \mathsf{f}(\mathsf{b})) = \mathsf{g}'(\mathsf{h}(\mathsf{a}, \mathsf{a}), \mathsf{f}(\mathsf{b})) \tag{7.3}$$

It appears this cannot be oriented left-to-right because $\succ^*$ would require $\mathsf{S}(\mathsf{a}) \succ \mathsf{h}(\mathsf{a}, \mathsf{a})$. Supporting this is the observation that if we have the rule $\mathsf{h}(\mathsf{a}, \mathsf{a}) \to \mathsf{a}$, then rewriting gives

$$\mathsf{g}(\mathsf{S}(\mathsf{a}), \mathsf{f}(\mathsf{b})) = \mathsf{g}'(\mathsf{a}, \mathsf{f}(\mathsf{b}))$$

and this is orientable because $\mathsf{S(a)} \succ \mathsf{a}$ and $\mathsf{g(S(a), f(b))} \succ \mathsf{f(b)}$. But if we have the rule $\mathsf{g'(h(x,x), y)} \rightarrow \mathsf{h(x,y)}$ instead, then rewriting gives

$$\mathsf{g(S(a), f(b))} = \mathsf{h(a, f(b))}$$

which is orientable because $\mathsf{g} > \mathsf{h}$. This suggests that it would be more accurate to say that (7.3) cannot be oriented because $\mathsf{g} \not> \mathsf{g'}$. Thus it is not always clear which subterms block an equation from being oriented.

However, it is not necessary to identify the precise set of subterms which need to be removed to allow orienting an equation. Using precedence violations to test for acceptability is a heuristic, so any inaccuracies can be ignored by the user. The important criteria is that if an equation cannot be oriented, then the set of precedence violations should be non-empty.

Definitions 7.3, 7.4, and 7.5 give mutually recursive functions computing precedence violations. The output is a multiset of pairs, $\{x_i \not> y_i\}$, such that x_i and y_i are either subterms or function symbols and x_i is not larger than y_i. Since a derivation might make progress by removing a precedence violation that is embedded within the context of another violation, the output includes violations from subterms even when they are embedded within other violations.

The definition of the primary function for computing precedence violations, PV, is given in Figure 7.1. Precedence violations for lexicographically ordered functions are defined as

Definition 7.4 Assume f, g with arity n are lexicographically ordered, $f \cong g$, and $f(\overline{u_n}) \not\succ g(\overline{v_n})$, and let k be the smallest number such that either $k > n$ or $u_1 \approx v_1, \ldots, u_{k-1} \approx v_{k-1}$, and $u_k \not\approx v_k$. By assumption, at least one of the following is false: $k \leq n$, $u_k \succ v_k$, $f(\overline{u_n}) \succ v_{k+1}$, $\ldots$, or $f(\overline{u_n}) \succ v_n$. Then

$$\begin{aligned} &PV\text{-lex}(f(\overline{u_n}), g(\overline{v_n})) = \\ &\qquad \begin{array}{ll} \{\langle \overline{u_n} \rangle \not> \langle \overline{v_n} \rangle\} & \text{if } k > n \\ PV(u_k, v_k) & \text{if } k \leq n \text{ and } u_k \not\succ v_k \\ \emptyset & \text{otherwise} \end{array} \end{aligned}$$

Note that the Definition 7.1 suggests that

$$\biguplus_{i \,\in\, k+1..n} PV(f(\overline{u_n}), v_i)$$

should be included in the multiset of violations in the second and third cases. However, these are already included as part of Ψ in the definition of PV, so including them in PV-lex would

Definition 7.3 If $f(\overline{u_n}) \succ g(\overline{v_m})$,

$$PV(f(\overline{u_n}), g(\overline{v_m})) = \emptyset$$

otherwise,

$$PV(f(\overline{u_n}), g(\overline{v_m})) =$$

$$\begin{array}{ll}
\{f \not\succ g(\overline{v_m})\} & \text{if } f \text{ is a variable} \\
\{f(\overline{u_n}) \not\succ g\} & \text{if } f \text{ is a function, } g \text{ is a variable,} \\ & \quad \text{and } g \notin \mathcal{FV}(f(\overline{u_n})) \\
\{f \not\succ g(\overline{v_m})\} & \text{if } f \cong g \text{ and } n = 0 \\
PV\text{-lex}(f(\overline{u_n}), g(\overline{v_m})) \uplus \Psi & \text{if } f \cong g,\ n = m, \text{ and both are} \\ & \quad \text{lexicographically ordered} \\
PV\text{-multi}(\{\overline{u_n}\}, \{\overline{v_m}\}) \uplus \Psi & \text{if } f \cong g \text{ and neither is} \\ & \quad \text{lexicographically ordered} \\
\Psi & \text{if } f > g \\
\{f \not\succ g\} \uplus \Psi & \text{otherwise}
\end{array}$$

where

$$\Psi = \biguplus_{i \in 1..n} PV(f(\overline{u_n}), v_i)$$

Figure 7.1: Definition of PV.

be redundant. Thus PV-lex(s, t) is empty unless all of the arguments are equivalent or the first $k-1$ arguments are equivalent and $u_k \not\succ v_k$.

For multisets, we first delete all subterms that are shared by both expressions and then examine the remaining subterms to find those pairs which are inconsistent with $\succ$. As a special case, we use the original sets if they are equal. This is to handle equations such as $f(\overline{u}) = g(\overline{u})$ where $f \cong g$.

Definition 7.5 Assume U and V are multisets of terms such that $U \not\succ\!\!\succ V$. Then if $U' = U \setminus V$ and $V' = V \setminus U$, we define

$$PV\text{-multi}\,(U, V) =$$

$$\begin{array}{ll}
\{U \not\succ V\} & \text{if } U' = V' = \emptyset \\
\{U' \not\succ \{v \in V' \mid \forall u \in U',\ u \not\succ v\}\} & \text{otherwise}
\end{array}$$

This first example is based on the derivation of $\mathsf{flatten}$ in Figure 3.3 (Section 3.1).

Example 7.6 In the derivation for $\mathsf{flatten}$, case analysis gives the equation

$$\mathsf{flatten}(\mathsf{Tree}(\mathsf{l},\mathsf{r}),\mathsf{a}) = \mathsf{append}(\mathsf{append}(\mathsf{fringe}(\mathsf{l}),\mathsf{fringe}(\mathsf{r})),\mathsf{a})$$

After executing `simplify()` (which has no effect on the equation), Focus computes the precedence violations and stores them in the script. Assume $\mathsf{append} > \mathsf{flatten}$, $\mathsf{fringe} > \mathsf{flatten}$, and $\mathsf{flatten}$ is lexicographically ordered from left to right. Then the precedence violations for this equation are

$$\begin{aligned}
&PV(\mathsf{flatten}(\mathsf{Tree}(\mathsf{l},\mathsf{r}),\mathsf{a}), \mathsf{append}(\mathsf{append}(\mathsf{fringe}(\mathsf{l}),\mathsf{fringe}(\mathsf{r})),\mathsf{a}))\\
&\quad= \{\mathsf{flatten} \not> \mathsf{append}\} \uplus PV(\mathsf{flatten}(\mathsf{Tree}(\mathsf{l},\mathsf{r}),\mathsf{a}), \mathsf{append}(\mathsf{fringe}(\mathsf{l}),\mathsf{fringe}(\mathsf{r})))\\
&\quad= \{\mathsf{flatten} \not> \mathsf{append},\ \mathsf{flatten} \not> \mathsf{append}\} \uplus PV(\mathsf{flatten}(\mathsf{Tree}(\mathsf{l},\mathsf{r}),\mathsf{a}), \mathsf{fringe}(\mathsf{l}))\\
&\qquad\ \uplus PV(\mathsf{flatten}(\mathsf{Tree}(\mathsf{l},\mathsf{r}),\mathsf{a}), \mathsf{fringe}(\mathsf{r}))\\
&\quad= \{\mathsf{flatten} \not> \mathsf{append},\ \mathsf{flatten} \not> \mathsf{append},\ \mathsf{flatten} \not> \mathsf{fringe},\ \mathsf{flatten} \not> \mathsf{fringe}\}
\end{aligned}$$

Rewriting append gives the final form of the recursive case:

$$\mathsf{flatten}(\mathsf{Tree}(\mathsf{l},\mathsf{r}),\mathsf{a}) = \mathsf{flatten}(\mathsf{l},\mathsf{flatten}(\mathsf{r},\mathsf{a}))$$

Because $\mathsf{flatten}$ is ordered lexicographically, there are no precedence violations for this equation. This reflects the goal that the new program $\mathsf{flatten}$ not depend on the specification of fringe and not contain any calls to append. All of this information is captured in the script for the second case of the $\mathsf{flatten}$ derivation:

```
simplify()
    {flatten ≯ append, flatten ≯ append,
    flatten ≯ fringe, flatten ≯ fringe}
rewrite(append)
```

The next set of examples illustrates PV-lex and PV-multi:

Examples 7.7 Assume the ordering on function symbols includes $\mathsf{pow} > * > \mathsf{fastpow}$. Then we have the following:

1. $PV(\mathsf{fastpow}(\mathsf{x},\mathsf{n},\mathsf{a}), \mathsf{pow}(\mathsf{x},\mathsf{n}) * \mathsf{a}) = \{\mathsf{fastpow} \not> *,\ \mathsf{fastpow} \not> \mathsf{pow}\}$

2. If $\mathsf{fastpow}$ is not lexicographically ordered, then

$$PV(\mathsf{fastpow}(\mathsf{x},\mathsf{S}(\mathsf{y}),\mathsf{a}), \mathsf{fastpow}(\mathsf{x},\mathsf{y},\mathsf{x} * \mathsf{a}))$$

$$\begin{aligned} &= PV\text{-multi}\,(\{\mathsf{x}, \mathsf{S}(\mathsf{y}), \mathsf{a}\}, \{\mathsf{x}, \mathsf{y}, \mathsf{x} * \mathsf{a}\}) \\ &\qquad \uplus\, PV(\mathsf{fastpow}(\mathsf{x}, \mathsf{S}(\mathsf{y}), \mathsf{a}), \mathsf{x} * \mathsf{a}) \\ &= \{\{\mathsf{S}(\mathsf{y}), \mathsf{a}\} \not> \{\mathsf{x} * \mathsf{a}\}\} \uplus \{\mathsf{fastpow} \not> *\} \end{aligned}$$

3. If $\mathsf{f} > \mathsf{g} \cong \mathsf{g}' > \mathsf{h}$ and g and g' are lexicographically ordered, then

$$\begin{aligned} &PV(\mathsf{g}(\mathsf{S}(\mathsf{a}), \mathsf{h}(\mathsf{b})), \mathsf{g}'(\mathsf{f}(\mathsf{a}, \mathsf{a}), \mathsf{h}(\mathsf{b}))) \\ &\quad = PV\text{-lex}(\mathsf{g}(\mathsf{S}(\mathsf{a}), \mathsf{h}(\mathsf{b})), \mathsf{g}'(\mathsf{f}(\mathsf{a}, \mathsf{a}), \mathsf{h}(\mathsf{b}))) \\ &\qquad\quad \uplus\, PV(\mathsf{g}(\mathsf{S}(\mathsf{a}), \mathsf{h}(\mathsf{b})), \mathsf{f}(\mathsf{a}, \mathsf{a})) \uplus\, PV(\mathsf{g}(\mathsf{S}(\mathsf{a}), \mathsf{h}(\mathsf{b})), \mathsf{h}(\mathsf{b})) \\ &\quad = \quad PV(\mathsf{S}(\mathsf{a}), \mathsf{f}(\mathsf{a}, \mathsf{a})) \uplus \emptyset \uplus \emptyset \\ &\quad = \quad \{\mathsf{S} \not> \mathsf{f}\} \end{aligned}$$

Finally, we apply precedence violations to the fastpow derivation.

Example 7.8 Consider the derivation of fastpow in Figure 6.6a. To orient the inductive hypothesis,

$$\mathsf{pow}(\mathsf{x}, \mathsf{n}) * \mathsf{a} \rightarrow \mathsf{fastpow}(\mathsf{x}, \mathsf{n}, \mathsf{a}) \tag{7.4}$$

we must have $\mathsf{pow} > * > \mathsf{fastpow}$.[4] Because the final result is tail-recursive, we also specify that fastpow is lexicographically ordered.

The initial state of case 2 is

$$\mathsf{fastpow}(\mathsf{x}, \mathsf{S}(\mathsf{y}), \mathsf{a}) = \mathsf{x} * \mathsf{pow}(\mathsf{x}, \mathsf{y}) * \mathsf{a}$$

This equation has the violations

$$\{\mathsf{fastpow} \not> *, \mathsf{fastpow} \not> *, \mathsf{fastpow} \not> \mathsf{pow}\}$$

Rewriting $*$ and picking the best choice gives

$$\mathsf{fastpow}(\mathsf{x}, \mathsf{S}(\mathsf{y}), \mathsf{a}) = \mathsf{fastpow}(\mathsf{x}, \mathsf{y}, \mathsf{x} * \mathsf{a}) \tag{7.5}$$

The remaining precedence violations for this equation are

$$\begin{aligned} &PV(\mathsf{fastpow}(\mathsf{x}, \mathsf{S}(\mathsf{y}), \mathsf{a}), \mathsf{fastpow}(\mathsf{x}, \mathsf{y}, \mathsf{x} * \mathsf{a})) \\ &\quad = PV\text{-lex}(\mathsf{fastpow}(\mathsf{x}, \mathsf{S}(\mathsf{y}), \mathsf{a}), \mathsf{fastpow}(\mathsf{x}, \mathsf{y}, \mathsf{x} * \mathsf{a})) \end{aligned}$$

[4]The ordering `pow` $> *$ is required by the definition of `pow`.

$$\begin{aligned} & \uplus\, PV(\mathsf{fastpow}(\mathsf{x}, \mathsf{S}(\mathsf{y}), \mathsf{a}), \mathsf{x} * \mathsf{a}) \\ = \; & \emptyset \uplus \{\mathsf{fastpow} \not> *\} \end{aligned}$$

In this case, rewriting achieved the goal of removing all occurrences of pow and one occurrence of $*$.

The remaining violation in Example 7.8 reflects an inadequacy in the lexicographic recursive path ordering. This is considered in the next section which discusses applying precedence violations to replay.

7.3 Using Violations in Replay

Because the term ordering is a user-specified preference relation, it can be used to determine the goals of a derivation. This section considers using information from precedence violations to infer goals and applying that information to a new problem to test for acceptability.

After Focus applies an operation, it computes the precedence violations for the resulting equation and stores them in the script-entry for the operation. After reapplying operations during replay, ReFocus verifies that the new set of precedence violations is a subset of the old violations. To allow for differences between specifications, ReFocus applies the visible transforms to the old violations before comparing sets. At the end of replaying a focus node, if the remaining violations are still not a subset of the original violations, ReFocus warns the user that the final results may not be acceptable and notes which operation was the first to fail to remove the necessary violations. The user can then examine the results more closely to determine why replay failed.

Note that this method allows the results of a derivation to contain precedence violations. Ideally, programs would contain no violations. However, it is not possible to require that all violations be resolved. One reason is that the user may not want to take the time to ensure that every rule can be oriented. This may result in a system that does not always terminate, but Focus allows such unsafe practices for convenience. But more importantly, the lexicographic recursive path ordering, as any ordering, is incomplete. That is, there are sets of rewrite rules which terminate but not for any definition of $>$. Ordering terms is undecidable (*cf.* [DJ90]), so requiring that a derivation remove all precedence violations is too restrictive because it does not allow for termination proofs constructed by the user outside of the scope of Focus.

This following examples illustrate using precedence violations to check for acceptability. First, we continue Example 4.1 (p. 36).

Example 7.9 In Example 4.1, the derivation of flatten is used to guide the derivation for a program squash. Case analysis results in the equation

$$\mathsf{squash}(\mathsf{Node}(\mathsf{l},\mathsf{i},\mathsf{r}),\mathsf{a}) = \mathsf{append}(\mathsf{append}(\mathsf{nodes}(\mathsf{l}),\mathsf{i}.\mathsf{nodes}(\mathsf{r})),\mathsf{a}) \tag{7.6}$$

Assuming append $>$ squash, nodes $>$ squash, and squash is lexicographically ordered from left to right, the violations for this equation are

$$\{\mathsf{squash} \not> \mathsf{append}, \mathsf{squash} \not> \mathsf{append}, \\ \mathsf{squash} \not> \mathsf{nodes}, \mathsf{squash} \not> \mathsf{nodes}\}$$

Rewriting equation (7.6) gives

$$\mathsf{squash}(\mathsf{Node}(\mathsf{l},\mathsf{i},\mathsf{r}),\mathsf{accum}) = \mathsf{squash}(\mathsf{l},\mathsf{i}.\mathsf{squash}(\mathsf{r},\mathsf{accum}))$$

As in Example 7.6, this equation contains no precedence violations, so ReFocus accepts the new result.

This is useful because transforming the original result gives

$$\mathsf{squash}(\mathsf{Node}(\mathsf{l},\mathsf{i},\mathsf{r}),\mathsf{a}) = \mathsf{squash}(\mathsf{l},\mathsf{squash}(\mathsf{r},\mathsf{a}))$$

While this is similar to the actual result, it is not identical. Using precedence violations allows ReFocus to accept the new result without complaint.

Example 6.22, in which the derivation for fastpow was used to guide the derivation for rzip, is similar to Example 7.9 in that the resulting program is accepted because there are no violations. However, this case differs in that fastpow is an example for which the lexicographic recursive path ordering is inadequate. To orient (7.4), $*$ must be larger than fastpow. But then, as shown in Example 7.8, the final equation contains the precedence violation $\{\mathsf{fastpow} \not> *\}$. This does not affect the derivation of rzip since each result contains no precedence violations. However, this is not always the case:

Example 7.10 Let mult be a function which multiples negative numbers as well as positive numbers:

```
mult(0, y)                  → 0
mult(Int(s, x),  0)         → 0
mult(Int(P, x), Int(P, y))  → Int(P, x * y)
mult(Int(P, x), Int(N, y))  → Int(N, x * y)
mult(Int(N, x), Int(P, y))  → Int(N, x * y)
mult(Int(N, x), Int(N, y))  → Int(P, x * y)
```

Also, modify pow to use the new function instead of $*$:

```
pow(i, 0)    → Int(P, 1)
pow(i, S(x)) → mult(i, pow(i, x))
```

This version takes an integer for the first argument and a natural number for the second. The new specification for fastpow is

$$\mathsf{fastpow}(\mathsf{i},\mathsf{n},\mathsf{a}) = \mathsf{mult}(\mathsf{pow}(\mathsf{i},\mathsf{n}),\mathsf{a})$$

Replaying the derivation for fastpow gives the final result

$$\mathsf{fastpow}(\mathsf{i},\mathsf{S}(\mathsf{x}),\mathsf{a}) = \mathsf{fastpow}(\mathsf{i},\mathsf{x},\mathsf{mult}(\mathsf{i},\mathsf{a}))$$

The precedence violation for this is

$$\{\mathsf{fastpow} \not\succ \mathsf{mult}\} \tag{7.7}$$

Comparing the initial specifications for both derivations gives the transform

$$\{* \Rightarrow \mathsf{mult}\}$$

Applying this to the original set of violations, $\{\mathsf{fastpow} \not\succ *\}$, gives (7.7). This allows ReFocus to accept the new program even though it contains precedence violations.

Example 7.11 Consider Example 6.23 in which we attempted to use the derivation of aprev to guide a derivation of fastpow. Because the problems contain few common symbols and the functions have different numbers of arguments, ReFocus was unable to use generalization to select the correct result. In lieu of other information, ReFocus randomly selects between alternatives (ii) and (iii). By chance, it picks (ii):

$$\mathsf{fastpow}(\mathsf{i},\mathsf{S}(\mathsf{y}),\mathsf{a}) = \mathsf{fastpow}(\mathsf{i},\mathsf{y},\mathsf{i}) * \mathsf{a} \tag{7.8}$$

This is the wrong alternative to pick because it is not tail-recursive.

Fortunately, the new result does not pass the check for precedence violations. The precedence violations for (7.8) are

$$\{\mathsf{fastpow} \not\succ *\} \uplus PV\text{-lex}(\mathsf{fastpow}(\mathsf{i},\mathsf{S}(\mathsf{y}),\mathsf{a}),\mathsf{fastpow}(\mathsf{i},\mathsf{y},\mathsf{i})) = \{\mathsf{fastpow} \not\succ *\} \uplus \emptyset$$

Since the recursive case aprev,

$$\mathsf{aprev}(\mathsf{a1.x},\mathsf{a}) = \mathsf{aprev}(\mathsf{x},\mathsf{a1.a})$$

has no precedence violations, ReFocus warns the user that the new derivation may not be acceptable.

These examples show that precedence violations provide a useful, simple test for acceptability in transformational implementation systems. In the first two cases, precedence violations suppress warning messages that might have occurred had we required an exact match. In the third case, precedence violations suggest that the new program may not be acceptable and note that the problem lies in the topmost position.

However, using precedence violations as the acceptance criteria can fail. One reason is that the user might be expecting the violation and not look at the results closely. Because $* > \mathsf{fastpow}$, both alternatives in Example 6.23 contain the precedence violation $\mathsf{fastpow} \not> *$. Thus the user might be tempted to ignore the warnings from ReFocus. But a more important failure is that comparing precedence violations may mistakenly lead to accepting programs which are in the wrong form:

Example 7.12 Suppose the derivation for $\mathsf{fastpow}$ is used to construct a derivation for aprev and ReFocus mistakenly chooses the result

$$\mathsf{apprev}(\mathsf{a1.x}, \mathsf{a}) = \mathsf{aprev}(\mathsf{x}, \mathsf{a1.Nil}) \mathbin{+\!\!+} \mathsf{a}$$

This has the precedence violation $\{\mathsf{aprev} \not> \mathbin{+\!\!+}\}$. If generalization produces the transforms $\{\mathsf{fastpow} \Rightarrow \mathsf{aprev}, * \Rightarrow \mathbin{+\!\!+}\}$, then the new precedence violations will be a subset of the transformed violations in the prototype, leading ReFocus to accept the results quietly even though the program is not tail-recursive.

The solution is to strengthen the term ordering of ReFocus so that derived programs do not contain precedence violations. Also, in a production environment, the user would need to examine any precedence violations in the final program once development is finished to ensure that the program is efficient enough. The primary usefulness of the check for precedence violations during replay is to allow the user to ignore small difference in programs while correcting errors or trying out different design decisions. Comparing precedence violations is not accurate enough to allow the user to replay derivations without ever looking at the results closely.

7.4 Conclusion

The obvious task for replay is to apply the steps from the prototype derivation to the new problem. Less obvious, but nearly as important, replay must check that the results of the new derivation are acceptable. A direct comparison is inadequate for any but the simplest examples. Instead, the test should attempt to recognize what was achieved in the prototype at a more abstract level and ensure that the new derivation achieves the same results.

This chapter has defined a test based on term orderings. Term orderings are an integral part of rewrite systems, ensuring that rewriting terminates and that inductive proofs are sound. For program derivation, they also express efficiency relationships: if $s \succ t$, then t is assumed to be more efficient than s. Initially, programs are specified as unoriented equations. Derivations make progress by simplifying equations until they can be oriented in the desired direction. By identifying which terms block orientation, we obtain a metric which can be used to compare results.

Examples show that this provides a useful test which is lenient enough that minor differences can be ignored but not so lenient that any program is accepted. The test can fail, but failures usually occur when derivations with precedence violations are used as prototypes. In any case, while the test is not always accurate and in a production environment the final version of a program would need to be examined manually, comparing precedence violations does free the user from checking and rechecking results during program development.

Chapter 8

ReFocus

The previous chapters have presented components needed to build a robust replay system. These components have been used to build a replay system for Focus, ReFocus. This chapter discusses ReFocus from a global viewpoint, briefly describing its implementation and how it is used. It also gives the results of applying ReFocus to a number of examples and evaluates its usefulness to program maintenance.

The examples comprise the bulk of the chapter. The first section reviews the examples in the preceding chapters. Whereas the preceding chapters presented portions of these examples to illustrate specific replay issues, this chapter presents the complete derivations. These are followed by examples which further illustrate the capabilities and limitations of ReFocus. Finally, ReFocus is applied to a pair of moderately-sized problems to demonstrate that using replay can save work for the user.

8.1 An Overview of ReFocus

This section gives an overview of ReFocus. We briefly describe its implementation and how the user applies it to a problem.

ReFocus is essentially an interpreter for derivation trees. Given a prototype derivation and a target node, ReFocus constructs a new derivation by reexecuting the commands used to create the prototype. Thus the primary input is a derivation and the primary output is a sequence of commands to be executed by Focus. This is illustrated in Figure 8.1. The main difference between ReFocus and an interpreter is that ReFocus modifies commands before executing them.

To use ReFocus, the user specifies the prototype derivation and target node and invokes replay. Specifying the prototype and target is done in either of two ways. In the first, the user moves the curser to a node and invokes the command `replay`. The current node becomes the prototype, and a new node is created to be the target. ReFocus then uses the initial state of the prototype to initialize the target, possibly after applying available transforms, and reexecutes

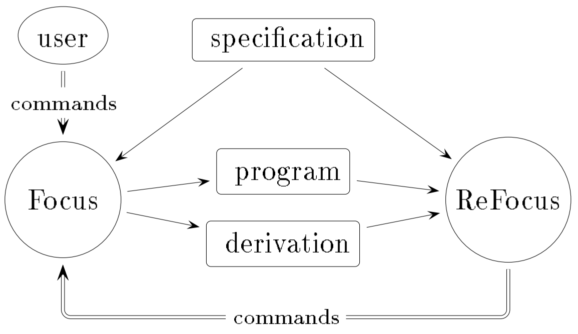


Figure 8.1: High-level view of ReFocus.

the derivation. Alternatively, the user marks a node to be the prototype, moves the cursor to the intended target, and invokes `replay-by-example`. ReFocus then reinitializes the target—without changing its contents—and reexecutes the derivation. The primary use of the first method is to reconstruct derivations after modifying specifications, while the primary use for the second is in derivation-by-analogy.

The implementation of ReFocus is based on a stack. Initially, the stack contains the root nodes of the target and prototype subtrees. ReFocus repeatedly retrieves the topmost pair of nodes from the stack, constructs transforms from the initial states and stores them in the derivation tree, and applies each operation in the prototype's script to the target. Before executing an operation, ReFocus applies any available transforms to its arguments. If the operation creates child nodes, such as `expand`, ReFocus uses generalization to pair the new children with the appropriate prototypes and pushes the pairs on to the stack. After finishing the script for a node, ReFocus compares the final states of the prototype and target and issues a warning if there are significant differences. Since ReFocus is organized around a stack, storing the current position in the script allows ReFocus to be interrupted and restarted by the user. This lets the user suspend ReFocus, repair a derivation manually (perhaps by introducing a new property or modifying the specification), and continue.

Like Focus, most of ReFocus is implemented in Emacs Lisp [LLtG90]. The complete system, including Focus, Tree-Mode,[1] and ReFocus, is approximately 41,300 lines of code.[2] Figure 8.2 illustrates the relationships between these systems and gives a rough approximation of the size of each component (in lines of code). In the figure, the nesting reflects how the system is layered; for instance, Focus calls Tree-Mode and GnuEmacs functions but not ReFocus functions.

[1] An Emacs Lisp implementation of Treemacs [Ham88] developed by Sam Kamin.
[2] Not counting blank lines and comments.

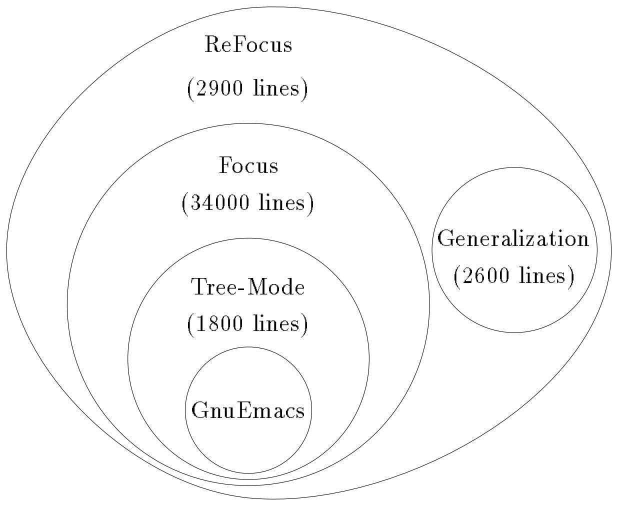


Figure 8.2: Components of the Focus system.

Generalization is shown as a discrete component of ReFocus. Approximately 1400 lines of this is a C++ program which computes the set of consistent matches $\mathcal{M}$ as defined in 6.13 (page 95). This code is written in C++ for speed because it represents the most computation-intensive part of the generalization algorithm.

8.2 Examples

The examples are in three groups: a review of previous examples, additional small examples, and two moderately-sized examples. For each, we give the specifications, prototype derivation, derivation created by replay, and key issues in executing replay. Except for minor differences, the given specifications can be input to Focus and the derivation trees are as they appear on the screen. The minor differences consist of font changes, trivial syntax changes (such as writing "*precedence relations:*" where one would write "`precedence`" in Focus), and joining short lines to make the examples fit on a single page. Any further edits are described in the examples.

8.3 Examples from Preceding Chapters

The first group consists of complete versions of the examples in the previous chapters.

precedence relations: {nodes, fringe} > append > {flatten, squash}
lexicographic append(1, 2), flatten(1, 2), squash(1, 2)

definitions:

```
    append(Nil, y) → y
    append(a.x, y) → a.append(x, y)

    fringe(Leaf(x)) → x.Nil
    fringe(Tree(left, right)) → append(fringe(left), fringe(right))

    nodes(Tip) → Nil
    nodes(Node(left, info, right)) → append(nodes(left), info.nodes(right))
```

properties:

```
    append(append(u, v), w) → append(u, append(v, w))
```

Figure 8.3: Specification of fringe and nodes.

8.3.1 Flattening Trees

Figure 8.3 gives the complete specifications for functions fringe and nodes, first introduced in Section 3.1, which return a list of items stored in a tree. The fringe function operates on trees with all the information stored in the leaves, while the nodes function operates on trees with the information stored in internal nodes. In addition, the specification gives the precedence relationships between function symbols (where {nodes, fringe} > append is equivalent to nodes > append and fringe > append), the lexicographic ordering on the arguments for all three functions, and the associative property for append. The associativity property, like all other properties listed in this chapter, can be proven using Focus. However, such proofs are omitted for brevity.

The fringe function makes two passes over the information stored in the tree. To improve it, we introduce an accumulating parameter:

```
flatten(tree, accum) = append(fringe(tree), accum)
```

We then derive a program for flatten by expanding fringe to obtain the cases tree = Leaf(x) and tree = Tree(left, right), rewriting each case, and picking the desired result in the second case. This derivation is given in Figure 8.4.

Other than minor differences described in the introduction, the tree shown in Figure 8.4 appears exactly as it does on the screen. Indentation expresses relationships between nodes; the children of a node are indented from their parent, with the root node at the top of the figure. The first line of each node contains the initial state, either as written by the user or as the result of case analysis. The second line shows that the node has been *closed*. This means

```
Focus: flatten(tree, accum) = append(fringe(tree), accum)
[closed with program:]
flatten(Leaf(x), accum) → x.accum
flatten(Tree(left, right), accum) → flatten(left, flatten(right, accum))
script:
  focus-on-spec(nil) ⇒ (see spec)
      {flatten ≯ append, flatten ≯ fringe}
  simplify(nil) [auto] ⇒ (no change)
  rewrite(nil) [compile, auto] ⇒ (no change)
  expand(fringe(tree)) [induction]

  > cases from fringe(tree)

    > case tree == Leaf(x)
    flatten(Leaf(x), accum) = append([x], accum)
    [closed]
    flatten(Leaf(x), accum) → x.accum
    script:
      simplify(nil) [auto] ⇒ flatten(Leaf(x), accum) → x.accum
      rewrite(nil) [compile, auto] ⇒ (no change)

    > case tree == Tree(left, right)
    flatten(Tree(left, right), accum) =
        append(append(fringe(left), fringe(right)), accum)
    [closed]
    flatten(Tree(left, right), accum) → flatten(left, flatten(right, accum))
    script:
      simplify(nil) [auto] ⇒ (no change)
          {flatten ≯ append, flatten ≯ append, flatten ≯ fringe, flatten ≯ fringe}
      rewrite(append) [choices == 2, compile] ⇒ (multiple results)
      pick-expr(flatten(Tree(left, right), accum) → flatten(left, flatten(right, accum)))
```

Figure 8.4: Derivation of flatten; prototype for derivation of squash.

that further operations on the node are disallowed.[3] The third line gives the final state of the node. For a top-level node, this includes the final program given by the derivation or the set of proven properties. The last section of the node gives the *script*; that is, the sequence of commands applied to the node. In addition to the commands and their arguments (where nil indicates no argument was given), the script contains details about how to replay each operation. These include the information given in square brackets (such as `[auto]`, which indicates Focus applied the operation automatically), intermediate results (marked by $\Rightarrow$), and any remaining precedence violations (shown a sets).

As discussed in Chapter 3, the derivation of flatten can be replayed on the specification of squash defined as

$$\mathsf{squash(tree,\ accum) = append(nodes(tree),\ accum)}$$

To take advantage of the procedures for associative matching built in to Focus, we also replace the associative property for append by the statement

declare associative append

Executing replay on the new specification gives the derivation shown in Figure 8.5. After completing replay, ReFocus prints the message

```
Replay... apparently succeeded
```

to indicate that it finds that the results of replay are acceptable.

ReFocus addresses four key issues in applying the derivation of flatten to squash:

- finding the appropriate expression to be expanded (nodes(tree)),
- matching the cases so rewrite(append) was applied to the appropriate case,
- picking the right result in the second case, and
- checking that the program for squash is acceptable.

The details for these steps are given in Examples 4.1, 5.51, 7.6, and 7.9.

To complete the derivation, we derive new versions of fringe and nodes which call flatten and squash with the appropriate parameters. The resulting derivations are shown in Figure 8.6.[4] In the figure, the notation [x] is an abbreviation for x.Nil; in general, the list $x_1.\cdots.x_n$.Nil is written as $[x_1, \cdots, x_n]$. The derivation of programs for fringe and nodes is straightforward and does not create significant issues for replay. Because this is almost always the case, we omit this final step from the remaining examples.

[3] For space, the [*closed*] marker is omitted in most of the derivation trees in this chapter, but all derivations are shown in the closed state.

[4] In this and subsequent derivations, details about commands and their results have been omitted to reduce clutter.

```
Focus: squash(tree, accum) = append(nodes(tree), accum)
[closed with program:]
squash(Tip, accum) → accum
squash(Node(left, info, right), accum) → squash(left, info.squash(right, accum))
script:
  focus-on-spec(nil) ⇒ (see spec) {squash ⊁ append, squash ⊁ nodes}
  simplify(nil) [auto] ⇒ (no change)
  rewrite(nil) [compile, auto] ⇒ (no change)
  expand(nodes(tree)) [induction]

  > cases from nodes(tree)

    > case tree == Tip
    squash(Tip, accum) = append(Nil, accum)
    [closed]
    squash(Tip, accum) → accum
    script:
      simplify(nil) [auto] ⇒ squash(Tip, accum) → accum
      rewrite(nil) [compile, auto] ⇒ (no change)

    > case tree == Node(left, info, right)
    squash(Node(left, info, right), accum) =
        append(append(nodes(left), info.nodes(right)), accum)
    [closed]
    squash(Node(left, info, right), accum) → squash(left, info.squash(right, accum))
    script:
      simplify(nil) [auto] ⇒ (no change)
          {squash ⊁ append, squash ⊁ append, squash ⊁ nodes, squash ⊁ nodes}
      rewrite(append) [choices == 2, compile] ⇒ (multiple results)
      pick-expr(squash(Node(left, info, right), accum) →
          squash(left, info.squash(right, accum)))

Replay... apparently succeeded
```

Figure 8.5: Result of replaying Figure 8.4 to derive squash.

Focus: fringe
[*closed with program:*]
fringe(Leaf(x)) → [x]
fringe(Tree(left, right)) → flatten(left, fringe(right))
script: `focus-on-function`

> \> fringe(Leaf(x)) = [x]
> fringe(Leaf(x)) → [x]
> *script:* `simplify, rewrite`
>
> \> fringe(Tree(left, right)) = append(fringe(left), fringe(right))
> fringe(Tree(left, right)) → flatten(left, fringe(right))
> *script:* `simplify, rewrite`

a. Derivation of program for fringe; prototype for (b).

Focus: nodes
[*closed with program:*]
nodes(Tip) → Nil
nodes(Node(left, info, right)) → squash(left, info.nodes(right))
script: `focus-on-function`

> \> nodes(Tip) = Nil
> nodes(Tip) → Nil
> *script:* `simplify, rewrite`
>
> \> nodes(Node(left, info, right)) = append(nodes(left), info.nodes(right))
> nodes(Node(left, info, right)) → squash(left, info.nodes(right))
> *script:* `simplify, rewrite`

```
Replay... apparently succeeded
```

b. Result of replaying derivation of fringe on nodes.

Figure 8.6: Derivation of programs for fringe and nodes.

8.3.2 Sorted Lists

Figure 8.7a specifies a quadratic-time predicate, sorted, which tests if a list is sorted from low to high. Elements in the list are compared by an arbitrary relation $<$. The statement

$$\textit{declare external}\ (<)$$

indicates that $<$ is intentionally undefined.[5] However, the property

$$\mathsf{x} < \mathsf{z}\ \{\mathsf{x} < \mathsf{y}\ \&\ \mathsf{y} < \mathsf{z}\} = \mathsf{True}$$

states that $<$ is assumed to be transitive, where $p\ \{q\} = r$ means $p = r$ whenever q is satisfied.

To derive a linear-time program for sorted, we start with the specification

$$\mathsf{sortedp(x)} = \mathsf{sorted(x)}$$

and expand twice. We then prove that if a is smaller than the first element of a sorted list, it is smaller than the entire list:

$$\mathsf{smaller(l,\ a)}\ \{\mathsf{a} < \mathsf{y}\ \&\ \mathsf{sortedp(y.l)}\} = \mathsf{True}$$

This property is then used to remove the redundant test in sortedp. The complete derivation is given in Figure 8.8, where $*$identity indicates that the given case is reduced to an identity of the form $u = u$.

To fit the derivation tree on a page and to reduce clutter, we have omitted initial states in nodes, *cases from* nodes (since the subterms that were expanded are given in the scripts in the parent nodes), and precedence violations. These are generally omitted in later derivations as well. The scripts in Figure 8.8 introduce two new operations: `add-inductive-hypotheses` and `close-subsidiary`. The first allows the user to force `expand` to introduce the inductive hypotheses, even when the given rule cannot be oriented. The second controls when a subsidiary focus node is replayed (relative to other operations in a script).

As described in Example 6.21, we can use replay to update the sortedp program after modifying sorted. Figure 8.7b gives the specification of a modified version of sorted which renames $<$ to less and adds a parameter dir. The dir parameter controls if the list is sorted from low to high or high to low. The specification includes a property stating that less is transitive, but this property can be proven using the transitivity of $<$. Replaying the derivation of sortedp after introducing less and dir gives the derivation in Figure 8.9. This figure omits a number of auxiliary properties created by the subsidiary proof which result from the relationship between $<$ and `less`.

Again, ReFocus constructs the new derivation without any error messages. The key to constructing the new derivation is updating the property focus to

[5] Writing an operator in parentheses allows it to be used in Focus as the name of a function rather than part of an expression.

```
precedence relations:
    sorted > smaller > sortedp > (<)
declare external (<)
definitions:
    smaller(Nil, x)   → True
    smaller(y.l, x)   → x < y & smaller(l, x)
    sorted(Nil)       → True
    sorted(a.l)       → smaller(l, a) & sorted(l)
properties:
    x < z {x < y & y < z} = True
```

a. Original specification.

```
precedence relations:
    sorted > smaller > sortedp > less > (<)
declare external (<)
definitions:
    less(a, b, Up)        → a < b
    less(a, b, Down)      → b < a
    smaller(x, Nil, dir)  → True
    smaller(x, y.l, dir)  → less(x, y, dir) & smaller(x, l, dir)

    sorted(Nil, dir)      → True
    sorted(a.l, dir)      → smaller(a, l, dir) & sorted(l, dir)
properties:
    x < z {x < y & y < z} = True
    less(x, z, dir) {less(x, y, dir) & less(y, z, dir)} = True
```

b. Modified specification.

Figure 8.7: Original and modified specifications of sorted.

```
Focus: sortedp(x) = sorted(x)
[closed with program:]
sortedp(Nil) → True
sortedp([a]) → sortedp(Nil)
sortedp(a.y.l1) → a < y & sortedp(y.l1)
script: simplify, rewrite, expand(sorted(x))

  > case x == Nil
  sortedp(Nil) → True
  script: simplify, rewrite

  > case x == a.l
  sortedp(a.l) = smaller(l, a) & sortedp(l)
  script: simplify, rewrite, expand(smaller(l,a)), close-subsidiary

    Prove: smaller(l, a) {a < y & sortedp(y.l)} = True
    [closed with properties:]
    smaller(l, a) {a < y & sortedp(y.l)} = True
    sortedp(y.l) {a < y & not(smaller(l, a))} = False
    a < y {not(smaller(l, a)) & sortedp(y.l)} = False
    script: simplify, rewrite, expand(smaller(l, a)), add-inductive-hypotheses

      > case l == Nil: *identity
      script: simplify, rewrite

      > case l == y1.l1
      a < y1 & smaller(l1, a) {a < y & sortedp(y.y1.l1)} = True
      script: universal(sortedp(y.l1))

        > case sortedp(y.l1)
        a < y1 {sortedp(y.l1) & a < y & sortedp(y.y1.l1)} = True
        script: simplify, rewrite, rewrite(smaller), simplify, expand(sortedp(y.y1.l1))

          > case True: *identity
          script: simplify, rewrite, pick-expr(a < y1 ...), rewrite(a < y1)

        > case not(sortedp(y.l1))
        a < y1 & smaller(l1, a) {not(sortedp(y.l1)) & a < y & sortedp(y.y1.l1)} = True
        script: simplify, rewrite, expand(sortedp(y.y1.l1))

          > case True: *identity
          script: simplify, rewrite

    > case l == Nil
    sortedp([a]) → sortedp(Nil)
    script: simplify, rewrite

    > case l == y.l1
    sortedp(a.y.l1) → a < y & sortedp(y.l1)
    script: simplify, rewrite(smaller), simplify
```

Figure 8.8: Derivation of original **sortedp**.

```
Focus: sortedp(x, dir) = sorted(x, dir)
[closed with program:]
sortedp(Nil, dir) → True
sortedp([a], dir) → sortedp(Nil, dir)
sortedp(a.y.l1, dir) → less(a, y, dir) & sortedp(y.l1, dir)
script: simplify, rewrite, expand(sorted(x, dir))

  > case x == Nil
  sortedp(Nil, dir) → True
  script: simplify, rewrite

  > case x == a.l
  sortedp(a.l, dir) = smaller(a, l, dir) & sortedp(l, dir)
  script: simplify, rewrite, expand(smaller(a,l,dir)), close-subsidiary

    Prove: smaller(a, l, dir) {less(a, y, dir) & sortedp(y.l, dir)} = True
    [closed with properties:]
    smaller(a, l, dir) {less(a, y, dir) & sortedp(y.l, dir)} = True
    ... derivative properties suppressed ...
    script: simplify, rewrite, expand(smaller(a, l, dir)), add-inductive-hypotheses

      > case l == Nil: *identity
      script: simplify, rewrite

      > case l == y1.l1
      less(a, y1, dir) & smaller(a, l1, dir) {less(a, y, dir) & sortedp(y.y1.l1, dir)} = True
      script: universal(sortedp(y.l1, dir))

        > case sortedp(y.l1, dir)
        less(a, y1, dir) {sortedp(y.l1, dir) & less(a, y, dir) & sortedp(y.y1.l1, dir)} = True
        script: simplify, rewrite, rewrite(smaller), simplify, expand(sortedp(y.y1.l1,dir))

          > case True: *identity
          script: simplify, rewrite, pick-expr(less(a,y1,dir) ...), rewrite(less(a,y1,dir))

        > case not(sortedp(y.l1, dir))
        less(a, y1, dir) & smaller(a, l1, dir)
            {not(sortedp(y.l1, dir)) & less(a, y, dir) & sortedp(y.y1.l1, dir)} = True
            script: simplify, rewrite, expand(sortedp(y.y1.l1, dir))

          > case True: *identity
          script: simplify, rewrite, pick-expr(*identity)

    > case l == Nil
    sortedp([a], dir) → sortedp(Nil, dir)
    script: simplify, rewrite

    > case l == y.l1
    sortedp(a.y.l1, dir) = less(a, y, dir) & sortedp(y.l1, dir)
    script: simplify, rewrite(smaller), simplify

Replay... apparently succeeded
```

Figure 8.9: Result of replaying Figure 8.8 after modifying sortedp.

smaller(l, a, dir) {less(a, y, dir) & sortedp(y.l, dir)} = True

ReFocus also updates a number of `expand`, `universal`, `rewrite`, and `pick-expr` operations to reflect the change from < to less. For details, see Example 6.21.

8.3.3 From Computing Exponentials to Joining Lists

Figure 8.10 gives the specification of a function pow(x,y) which raises x to the power of y. To make this function tail-recursive, we introduce an accumulating parameter a:

fastpow(x, n, a) = pow(x, n) $*$ a

As shown in Figure 8.10, this leads to a tail-recursive program for fastpow.

The derivation of fastpow can be used to guide the derivation of a program to join lists. Figure 8.11 specifies a function revzip which joins two lists in reverse order. As with fastpow, this function can be made tail-recursive by introducing an accumulating parameter:

rzip(u, v, a) = revzip(u, v) ++ a

Given this specification, replaying the derivation in Figure 8.10 gives the derivation in Figure 8.11. The new derivation is constructed with no error messages.

During replay, ReFocus updates `expand` and `rewrite` steps to reflect name-space changes. It also matches cases; this is particularly important in this example since there are more cases in the result than in the prototype, and both of the extra cases must be matched to the second case of fastpow. After rewriting ++, ReFocus also picks the correct result in each of the recursive cases. For details on these steps, see Example 6.22. Finally, ReFocus verifies that the new program is acceptable. In this case, it is acceptable because while the prototype has precedence violations, the new program does not.

Figure 8.12 specifies a version of pow which has been extended to allow negative values. Each number is represented by sign and magnitude, and functions have been introduced to add, subtract, and multiply such numbers. Replaying the derivation of fastpow in Figure 8.10 on the specification

fastpow(v, n, a) = mult(pow(i, n), a)

gives the derivation in Figure 8.13. As for the rzip derivation, ReFocus updates `rewrite` and `expand` steps, matches cases appropriately, and picks the correct result in the recursive case. Equally importantly, ReFocus accepts the new derivation without any error messages. As discussed in Example 7.10, the new program contains precedence violations, but these match the violations in the prototype (after applying transforms).

```
precedence relations: pow > (∗) > {(+), fastpow}
lexicographic fastpow(1, 2, 3)
declare ac (+), (∗)

definitions:
    a + 0     → a            a + S(b)      → S(a + b)
    a ∗ 0     → 0            a ∗ S(b)      → a + a ∗ b
    pow(x, 0) → 1            pow(x, S(y)) → x ∗ pow(x, y)


Focus: fastpow(x, n, a) = pow(x, n) ∗ a
[closed with program:]
fastpow(x, 0, a) → a
fastpow(x, S(y), a) → fastpow(x, y, x ∗ a)
script:
  focus-on-spec  {fastpow ≯ ∗, fastpow ≯ pow}
  simplify
  rewrite
  expand(pow(x, n))

  > cases from pow(x, n)

    > case n == 0
    fastpow(x, 0, a) = 1 ∗ a
    fastpow(x, 0, a) → a
    script:
      rewrite(∗)  {fastpow ≯ (+)}
      rewrite

    > case n == S(y)
    fastpow(x, S(y), a) = x ∗ pow(x, y) ∗ a
    fastpow(x, S(y), a) = fastpow(x, y, x ∗ a)
    script:
      rewrite(∗)
      pick-expr(fastpow(x, S(y), a) = fastpow(x, y, x ∗ a))  {fastpow ≯ ∗}
```

Figure 8.10: Specification and derivation of fastpow.

precedence relations: revzip $>$ (++) $>$ rzip
lexicographic 1 ++ 2, rzip(1, 2, 3)
declare associative (++)

definitions:

Nil ++ y	$\rightarrow$ y	revzip(a.as, Nil)	$\rightarrow$ revzip(as, Nil) ++ [a]
a.x ++ y	$\rightarrow$ a.(x ++ y)	revzip(Nil, b.bs)	$\rightarrow$ revzip(Nil, bs) ++ [b]
revzip(Nil, Nil)	$\rightarrow$ Nil	revzip(a.as, b.bs)	$\rightarrow$ revzip(as,bs) ++ [a,b]

Focus: rzip(u, v, a) = revzip(u, v) ++ a
[*closed with program*:]
rzip(Nil, Nil, a) $\rightarrow$ a
rzip(a1.as, Nil, a) $\rightarrow$ rzip(as, Nil, a1.a)
rzip(Nil, b.bs, a) $\rightarrow$ rzip(Nil, bs, b.a)
rzip(a1.as, b.bs, a) $\rightarrow$ rzip(as, bs, a1.b.a)
script:
  focus-on-spec {rzip $\not>$ ++ , rzip $\not>$ revzip}
  simplify, rewrite, expand(revzip(u, v))

  $>$ *cases from* revzip(u, v)

    $>$ *case* v == Nil & u == Nil
    rzip(Nil, Nil, a) = Nil ++ a
    rzip(Nil, Nil, a) $\rightarrow$ a
    script: rewrite(++)

    $>$ *case* v == Nil & u == a1.as
    rzip(a1.as, Nil, a) = revzip(as, Nil) ++ [a1] ++ a
    rzip(a1.as, Nil, a) $\rightarrow$ rzip(as, Nil, a1.a)
    script:
      rewrite(++), pick-expr(rzip(a1.as, Nil, a) $\rightarrow$ rzip(as, Nil, a1.a))

    $>$ *case* v == b.bs & u == Nil
    rzip(Nil, b.bs, a) = revzip(Nil, bs) ++ [b] ++ a
    rzip(Nil, b.bs, a) $\rightarrow$ rzip(Nil, bs, b.a)
    script:
      rewrite(++), pick-expr(rzip(Nil, b.bs, a) $\rightarrow$ rzip(Nil, bs, b.a))

    $>$ *case* v == b.bs & u == a1.as
    rzip(a1.as, b.bs, a) = revzip(as, bs) ++ [a1, b] ++ a
    rzip(a1.as, b.bs, a) $\rightarrow$ rzip(as, bs, a1.b.a)
    script:
      rewrite(++), pick-expr(rzip(a1.as, b.bs, a) $\rightarrow$ rzip(as, bs, a1.b.a))

```
Replay... apparently succeeded
```

Figure 8.11: Specification of rzip and result of replaying Figure 8.10.

precedence relations:

```
    add > (−) > {norm, sub}
    add > (+)
    pow > mult > {(∗), fastpow}
```

lexicographic fastpow(1, 2, 3)

declare ac mult

definitions:

```
    norm(0) → 0
    norm(Int(s, 0)) → 0
    norm(Int(s, S(x))) → Int(s, S(x))

    x − y → norm(sub(x, y))
    sub(0, 0) → 0
    sub(0, S(y)) → Int(N, S(y))
    sub(S(x), 0) → Int(P, S(x))
    sub(S(x), S(y)) → sub(x, y)

    add(0, y) → y
    add(Int(s, x), 0) → Int(s, x)
    add(Int(P, x), Int(P, y)) → Int(P, x + y)
    add(Int(N, x), Int(N, y)) → Int(N, x + y)
    add(Int(P, x), Int(N, y)) → x − y
    add(Int(N, x), Int(P, y)) → y − x

    mult(0, y) → 0
    mult(Int(s, x), 0) → 0
    mult(Int(P, x), Int(P, y)) → Int(P, x ∗ y)
    mult(Int(P, x), Int(N, y)) → Int(N, x ∗ y)
    mult(Int(N, x), Int(P, y)) → Int(N, x ∗ y)
    mult(Int(N, x), Int(N, y)) → Int(P, x ∗ y)

    pow(i, 0) → Int(P, 1)
    pow(i, S(x)) → mult(i, pow(i, x))
```

properties:

```
    mult(Int(P, 1), a) → a
```

Figure 8.12: Integer-based pow specification.

```
Focus: fastpow(i, n, a) = mult(pow(i, n), a)
[closed with program:]
fastpow(i, 0, a) → a
fastpow(i, S(x), a) → fastpow(i, x, mult(i, a))
script:
  focus-on-spec  {fastpow ⊁ mult, fastpow ⊁ pow}
  simplify
  rewrite
  expand(pow(i, n))

  > cases from pow(i, n)

    > case n == 0
    fastpow(i, 0, a) = mult(Int(P, 1), a)
    fastpow(i, 0, a) → a
    script:
      rewrite(mult)

    > case n == S(x)
    fastpow(i, S(x), a) = mult(mult(i, pow(i, x)), a)
    fastpow(i, S(x), a) = fastpow(i, x, mult(i, a))
    script:
      rewrite(mult)
      pick-expr(fastpow(i, S(x), a) = fastpow(i, x, mult(i, a)))
           {fastpow ⊁ mult}

Replay... apparently succeeded
```

Figure 8.13: Result of replaying Figure 8.10 on modified version of fastpow.

8.3.4 From Reversing Lists to Computing Exponentials

The final example from the preceding chapters illustrates ReFocus failing to construct the desired program but succeeding in warning the user of potential problems.

Figure 8.14a specifies a function, rev, which reverses a list. As shown, the specification

$$\mathsf{aprev}(\mathsf{l},\ \mathsf{a}) = \mathsf{rev}(\mathsf{l}) \mathbin{+\!\!+} \mathsf{a}$$

leads to a tail-recursive program for reversing lists. Replaying this derivation on the specification fastpow gives the derivation in Figure 8.14b. As discussed in Example 6.23, ReFocus chooses the wrong alternative after rewriting $*$ in the second case.

However, ReFocus does warn the user that the new derivation may not be acceptable. These messages are given in Figure 8.15. The warning in the first case arises because while `rewrite`($+\!\!+$) is sufficient in the aprev derivation, an extra `rewrite` step is needed in the fastpow derivation to completely reduce the term. The warning in the second case arises because the generalization algorithm failed. Since there are significant differences between the two problems in both the structure and function names, $+\!\!+$ was not transformed to $*$, and so `rewrite` failed to produce the desired result. The user must repair the derivation by applying the `rewrite`($+\!\!+$) step manually.[6]

Because ReFocus produced error messages, it prints

```
Replay... failed
```

after completing replay. While replaying aprev to derive fastpow fails, Section 8.4.1 shows that replay is successful on a related example based on computing factorials.

[6]Both of these problems could be avoided by modifying the Focus `rewrite` operation to always apply associativity and commutativity. However, this would require significant modifications to Focus.

```
precedence relation:
    rev > (++) > aprev
lexicographic 1 ++ 2, aprev(1, 2)
declare associative (++)
definitions:
    Nil ++ y → y
    a.x ++ y → a.(x ++ y)
    rev(Nil) → Nil
    rev(a.x) → rev(x) ++ [a]
```

```
Focus: aprev(l, a) = rev(l) ++ a
[closed with program:]
aprev(Nil, a) → a
aprev(a1.x, a) → aprev(x, a1.a)
script: simplify, rewrite, expand(rev(l))

  > cases from rev(l)

    > case l == Nil
    aprev(Nil, a) = Nil ++ a
    aprev(Nil, a) → a
    script: rewrite(++)

    > case l == a1.x
    aprev(a1.x, a) = rev(x) ++ [a1] ++ a
    aprev(a1.x, a) → aprev(x, a1.a)
    script: rewrite(++), pick-expr(...)
```

a. Specification and derivation of aprev.

```
Focus: fastpow(x, n, a) = pow(x, n) * a
[closed with program:]
fastpow(x, 0, a) → a + 0
fastpow(x, S(y), a) → fastpow(x, y, x) * a
script: simplify,rewrite,expand(pow(x, n))

  > cases from pow(x, n)

    > case n == 0
    fastpow(x, 0, a) = 1 * a
    fastpow(x, 0, a) = a + 0
    script: rewrite(*)  {fastpow ⊁ (+)}

    > case n == S(y)
    fastpow(x, S(y), a) = x * pow(x, y) * a
    fastpow(x, S(y), a) = fastpow(x, y, x) * a
    script: rewrite(*), pick-expr(...)

Replay... failed
```

b. Result of replaying (a) to derive fastpow.

Figure 8.14: Replaying aprev to derive fastpow.

```
Replay error #1 @ /4,1,1 ? case n == 0:
    Both initial and final states do not match prototype
    new: fastpow(x, 0, a) = a + 0
         violations: {fastpow ⊁ (+)}
    old: aprev(Nil, a) → a
         violations: none

Replay error #2 @ /4,1,2 ? case n == S(y):
    Final state does not match prototype
    new: fastpow(x, S(y), a) = fastpow(x, y, x) * a
         violations: {fastpow ⊁ (*)}
    old: aprev(a1.x, a) → aprev(x, a1.a)
         violations: none
```

Figure 8.15: Error messages from replaying the aprev to derive fastpow.

8.4 Illustrations of ReFocus's Strengths and Weaknesses

The preceding examples were designed to exhibit key aspects of replay: modifying derivations and testing results. In contrast, the examples in this section are designed to show some of the strengths and weaknesses of ReFocus as a complete system and to suggest possible improvements.

8.4.1 From Reversing a List to Computing Factorials

The first of these continues the example in Section 8.3.4. While the derivation of aprev is not useful as a prototype for deriving fastpow, it is useful as a prototype for deriving a tail-recursive version of a function which computes factorials, fact. Figure 8.16a defines fact, and Figure 8.16b shows the results of replaying the derivation of aprev on the specification

$$\mathsf{f}(\mathsf{x},\ \mathsf{a}) = \mathsf{fact}(\mathsf{x}) * \mathsf{a}$$

Again, ReFocus generates error messages. These are given in Figure 8.16c. But in this case, replay did not actually fail. The first message reflects the fact that an explicit rewrite of $*$ is needed to simplify the result. The second reflects the limitation of the term ordering used in Focus. As described in Section 7.3, the result

$$\mathsf{f}(\mathsf{S}(\mathsf{n}),\ \mathsf{a}) = \mathsf{f}(\mathsf{n},\ \mathsf{a} + \mathsf{a} * \mathsf{n})$$

is acceptable, but ReFocus generates an error because the prototype contains no violations while the new result does. But in spite of the error messages, ReFocus is successful in obtaining a tail-recursive program for f.

Discussion

ReFocus was successful on this example because both rev and fact have one parameter. This means that generalization can establish correspondences between f and aprev, $*$ and $+\!\!+$, and fact and rev. However, having the same number of parameters does not ensure replay will be successful. Consider the derivation in Figure 8.17a in which the arguments to $*$ have been reversed:

$$\mathsf{f}(\mathsf{x},\ \mathsf{a}) = \mathsf{a} * \mathsf{fact}(\mathsf{x})$$

ReFocus fails to obtain a tail-recursive program and prints the error message given in Figure 8.17b.

The problem is that generalization failed. Comparing the focus specifications for aprev and f gives the transform $x_1 +\!\!+\, x_2 \Rightarrow\ x_2 * x_1$ (among others). This transform is too specific because it only matches a term with arguments. As a result, `rewrite`$(*)$ is not executed in the

precedence relations:
    fact $>$ $(*)$ $>$ $(+)$ $>$ f
lexicographic f(1, 2)
declare ac $(+)$, $(*)$
definitions:

```
fact(0)     → 1
fact(S(n))  → S(n) * fact(n)
a + 0       → a
a + S(b)    → S(a + b)
a * 0       → 0
a * S(b)    → a + a * b
```

a. Specifications.

Focus: f(x, a) = fact(x) $*$ a
[*closed with program:*]
f(0, a) $\rightarrow$ 1 $*$ a
f(S(n), a) $\rightarrow$ f(n, a + a $*$ n)
script: `simplify, rewrite,`
  `expand(fact(x))`

  $>$ *cases from* fact(x)

    $>$ *case* x == 0
    f(0, a) = 1 $*$ a
    f(0, a) $\rightarrow$ 1 $*$ a
    script: `simplify, rewrite`

    $>$ *case* x == S(n)
    f(S(n), a) = S(n) $*$ fact(n) $*$ a
    f(S(n), a) = f(n, a + a $*$ n)
    script: `simplify, rewrite(*),`
      `pick-expr(...)`

```
Replay... failed
```

b. Results of replaying aprev (8.14a) to derive f.

```
Replay error #1 @ /4,1,1 ? case x == 0:
    Final state does not match prototype
    new: f(0, a) = 1 * a
         violations: {f ≯ (*)}
    old: aprev(Nil, a) → a
         violations: none

Replay error #2 @ /4,1,2 ? case x == S(n):
    Both initial and final states do not match prototype
    new: f(S(n), a) = f(n, a + a * n)
         violations: {f ≯ (*), f ≯ (+)}
    old: aprev(a1.x, a) → aprev(x, a1.a)
         violations: none
```

c. Error messages.

Figure 8.16: Replaying aprev to derive f.

```
Focus: f(x, a) = a ∗ fact(x)
[closed with program:]
f(0, a) → a
f(S(n), a) → a ∗ (S(n) ∗ fact(n))
script: simplify, rewrite,
  expand(fact(x))

  > cases from fact(x)

    > case x == 0
    f(0, a) = a ∗ 1
    f(0, a) → a
    script: simplify, rewrite

    > case x == S(n)
    f(S(n), a) = a ∗ (S(n) ∗ fact(n))
    f(S(n), a) = a ∗ (S(n) ∗ fact(n))
    script: simplify, rewrite

Replay... failed
```

a. Unsuccessful result of replaying aprev on modified f.

```
Replay error #1 @ /4,1,2 ? case x == S(n):
    Final state does not match prototype
    new: f(S(n), a) = a ∗ (S(n) ∗ fact(n))
         violations: {f ≯ (∗), f ≯ (∗), f ≯ fact}
    old: aprev(a1.x, a) → aprev(x, a1.a)
         violations: none
```

b. Error output.

Figure 8.17: Output of replaying aprev on alternative specification of f.

derivation of f. Thus generalization fails for the same reason that it fails in Example 6.23: we have simultaneously changed the names of functions and the structure of their arguments.

One solution would be to change how ReFocus applies generalization. Instead of attempting to simply transform the argument to the `rewrite` step, ReFocus should ensure that the argument does refer to a symbol which appears in the equation being acted upon,

$$f(S(n), a) = a * (S(n) * fact(n))$$

This could be done by using generalization to match the structure of this case of f against the corresponding case of aprev,

$$aprev(a1.x, a) = rev(x) +\!\!+ [a1] +\!\!+ a$$

ReFocus could then use this information to match $*$ with $+\!\!+$. This use of generalization would improve the robustness of replay. However, such applications of generalization must be defined more precisely before implementing them in ReFocus; this is left as future work.

```
precedence relations:
    sum > plus, app > add
lexicographic plus(1, 2)
definitions:
    plus(0, y)          → y
    plus(S(x), y)       →
          S(plus(x, y))
    sum(Empty)          → 0
    sum(Add(n, x))      →
          plus(n, sum(x))
    app(Empty, y)       → y
    app(Add(n, x), y)   →
          Add(n, app(x, y))
properties:
    plus(plus(x, y), z) →
          plus(x, plus(y, z))
```

a. Specification of sum.

```
Prove: sum(app(x, y)) →
          plus(sum(x), sum(y))
[closed with properties:]
sum(app(x, y)) → plus(sum(x), sum(y))
script: simplify,rewrite,expand(app(x, y))

  > cases from app(x, y)

    > case x == Empty
    sum(y) = plus(sum(Empty), sum(y))
    *identity
    script: simplify, rewrite

    > case x == Add(n, x1)
    sum(Add(n, app(x1, y))) →
          plus(sum(Add(n, x1)), sum(y))
    *identity
    script: simplify, rewrite
```

b. Proof.

Figure 8.18: Proof that sum distributes over app.

8.4.2 Examples Illustrating Large-Grained Operators

Because Focus operators are large-grained, many examples are trivial for replay. This section illustrates this using examples drawn from other work.

Examples from [KW94]

Kolbe and Walther [KW94] give several reuse examples in theorem proving. Their work, based on explanation-based learning [DM86, MKKC86], identifies key steps in proofs and generalizes them to construct proof schemata to be applied to other problems. However, such an analysis is not necessary in ReFocus (at least for the given examples) because Focus operators are large-grained.

Figure 8.18a defines a function, sum, which adds up the numbers in a list. The derivation in Figure 8.18b shows that this function distributes over the append operator app:[7]

sum(app(w, y)) = plus(sum(x), sum(y))

This derivation can be used to prove a related result for multiplication,

prod(app(x, y)) = times(prod(x), prod(y))

[7]We use the names from [KW94] for consistency with that paper.

where times multiplies two numbers and prod multiplies the numbers in a list. The specification and result of replaying Figure 8.18b are given in Figure 8.19a. ReFocus completes this proof without any error messages.

Similarly, replaying Figure 8.18b can be used to show that computing list length distributes over appending lists:

$$\mathsf{len}(\mathsf{app}(\mathsf{x}, \mathsf{y})) = \mathsf{plus}(\mathsf{len}(\mathsf{x}), \mathsf{len}(\mathsf{y}))$$

The specification of these functions and the result of replay are given in Figure 8.19b. Again, there are no error messages.

Discussion

These examples are simple for ReFocus because of the emphasis on large-grained operators in Focus. Replay would have been successful even if ReFocus did not support generalization and comparing precedence violations. Each proof consists of expanding app and applying the default `simplify` and `rewrite` operations. Generalization does not play a role since the same term is expanded in each and since each case has the same script. This demonstrates that large-grained operators simplify replay.

Example from HOL

Section 2.1 gives the HOL [Gor88] proofs for distributing len over app and times over plus. Because the proofs are so dissimilar, adding a replay system to HOL which could handle this example would be difficult. In contrast, the steps are the same for both proofs in Focus.

Given the definitions in Figures 8.18a and 8.19, replaying the derivation in Figure 8.19b gives a proof that times distributes over plus as shown in Figure 8.20. Again, ReFocus completes the proof without errors, and generalization does not play a role.

```
precedence relations:
    prod > times > plus
lexicographic times(1, 2)
definitions:
    times(0, y)        → 0
    times(S(x), y)     →
          plus(y, times(x, y))
    prod(Empty)        → S(0)
    prod(Add(n, x))    →
          times(n, prod(x))
properties:
    times(times(x, y), z) →
          times(x, times(y, z))
    plus(a, 0) → a
```

```
Prove: prod(app(x, y)) → times(prod(x), prod(y))
[closed with properties:]
prod(app(x, y)) → times(prod(x), prod(y))
script: simplify,rewrite,expand(app(x, y))

  > cases from app(x, y)

    > case x == Empty
    prod(y) = times(prod(Empty), prod(y))
    *identity
    script: simplify, rewrite

    > case x == Add(n, x1)
    prod(Add(n, app(x1, y))) →
        times(prod(Add(n, x1)), prod(y))
    *identity
    script: simplify, rewrite

Replay... apparently succeeded
```

a. Result of replaying Figure 8.18b to show that prod distributes over app.

```
precedence relations:
    len > plus
definitions:
    len(Empty)      → 0
    len(Add(n, x)) → S(len(x))
properties:
    plus(plus(x, y), z) →
        plus(x, plus(y, z))
```

```
Prove: len(app(x, y)) → plus(len(x), len(y))
[closed with properties:]
len(app(x, y)) → plus(len(x), len(y))
script: simplify,rewrite,expand(app(x, y))

  > cases from app(x, y)

    > case x == Empty
    len(y) = plus(len(Empty), len(y))
    *identity
    script: simplify, rewrite

    > case x == Add(n, x1)
    len(Add(n, app(x1, y))) →
        plus(len(Add(n, x1)), len(y))
    *identity
    script: simplify, rewrite

Replay... apparently succeeded
```

b. Result of replaying Figure 8.18b to show that len distributes over app.

Figure 8.19: Replay examples from [KW94].

Prove: times(plus(a, b), c) $\rightarrow$ plus(times(a, c), times(b, c))
[*closed with properties*:]
times(plus(a, b), c) $\rightarrow$ plus(times(a, c), times(b, c))
script: `simplify, rewrite, expand(plus(a, b))`

> $>$ *cases from* plus(a, b)
>
> > $>$ *case* a == 0
> > times(b, c) = plus(times(0, c), times(b, c))
> > $*$identity
> > *script:* `simplify, rewrite`
> >
> > $>$ *case*a == S(x)
> > times(S(plus(x, b)), c) $\rightarrow$ plus(times(S(x), c), times(b, c))
> > $*$identity
> > *script:* `simplify, rewrite`

```
Replay... apparently succeeded
```

Figure 8.20: Result of replaying Figure 8.19b to show that times distributes over plus.

8.5 Moderately-sized Examples

Whereas the previous examples were designed to illustrate ReFocus, the remaining examples in this chapter are designed to show the usefulness of replay by applying it to moderately-sized examples. These examples are useful because they involve a number of steps and so show that replay can save the user a significant amount of work.

8.5.1 Circuit Verification

The following example, motivated by [GJCOG89], is based on verifying a circuit which computes the parity of a bitstream. Figure 8.21 defines two parity functions: oddParity, which returns 0 if a bitstream contains an odd number of 1's and 1 otherwise, and evenParity, which returns 0 if a bitstream contains an even number of 1's and 1 otherwise.[8]

[8] This specification introduces *precondition* statements to constrain how the functions are called. In this case, the preconditions specify types. Type declarations could be used instead, but these are currently only supported for documentation; they are not applied anywhere by Focus.

```
precedence relations:
    {evenParity, oddParity} > {all_binary, neg} > binary

precondition oddParity(l) : l != Nil & all_binary(l)
precondition evenParity(l) : l != Nil & all_binary(l)
precondition neg(i) : binary(i)

definitions:
    binary(x) → x == 0 | x == 1
    all_binary(Nil) → True
    all_binary(i.input) → binary(i) & all_binary(input)
    neg(1) → 0
    neg(0) → 1

    oddParity([0]) → 1
    oddParity([1]) → 0
    oddParity(1.rest) {rest != Nil} → neg(oddParity(rest))
    oddParity(0.rest) {rest != Nil} → oddParity(rest)

    evenParity([0]) → 0
    evenParity([1]) → 1
    evenParity(0.rest) {rest != Nil} → evenParity(rest)
    evenParity(1.rest) {rest != Nil} → neg(evenParity(rest))
```

Figure 8.21: Parity functions.

The circuit in Figure 8.22a (reproduced from [GJCOG89] with minor modifications) is intended to implement oddParity, while the circuit in Figure 8.22b is intended to implement evenParity. The devices used in the circuits are

`NEG`:	negates its input.
`ZERO`:	continuously produces 0.
`ONE`:	continuously produces 1.
`MUX`:	uses the leftmost input line to control which of the remaining inputs is passed through, where 0 selects the rightmost line.
`REG`:	saves its input when clocked and outputs that value until the next cycle.

The definitions in Figure 8.23 encode the odd parity circuit. The state of the circuit is represented by State($r1, r2, out$) where $r1$ is the state of `REG 1`, $r2$ is the state of `REG 2`, and out is the circuit's output. The top-level function is oParCirc which initializes the circuit and calls opar. The function opar models each iteration of the sircuity by feeding the register values and input into the circuit to compute the new output value.

To prove the circuit's correctness, we show

oddParity(inp) {all_binary(inp) & inp != Nil} $\rightarrow$ output(oParCirc(inp))

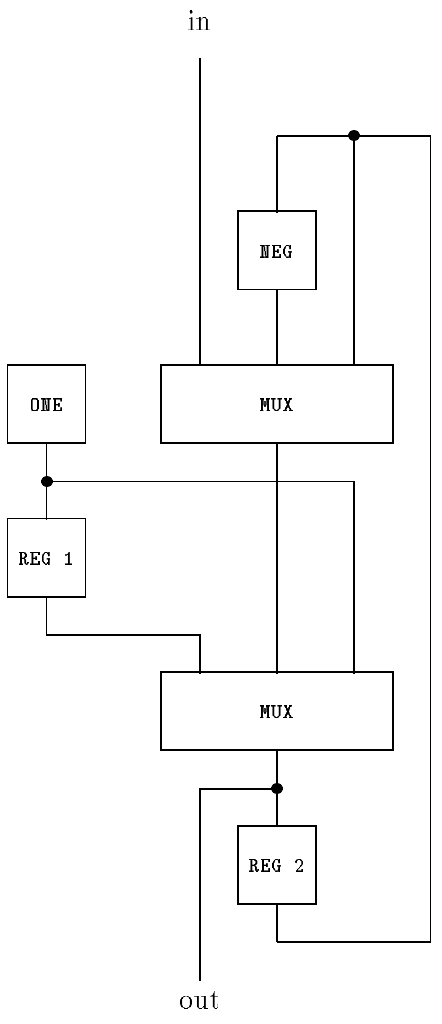
The proof is shown in Figures 8.24 and 8.25. It is constructed by repeatedly expanding opar and mux until each case is an instance of an earlier case.[9] Then rewriting is applied to reduce the case to *identity.

This proof can be used to help verify the even parity circuit in Figure 8.22b. Figure 8.26 specifies functions eParCirc and epar which implement this circuit. The key differences between these functions and oParCirc and opar are marked with boxes: the device `ONE` has been replaced by `ZERO` and the inputs to the lower `MUX` device have been swapped. To verify the resulting circuit, we prove

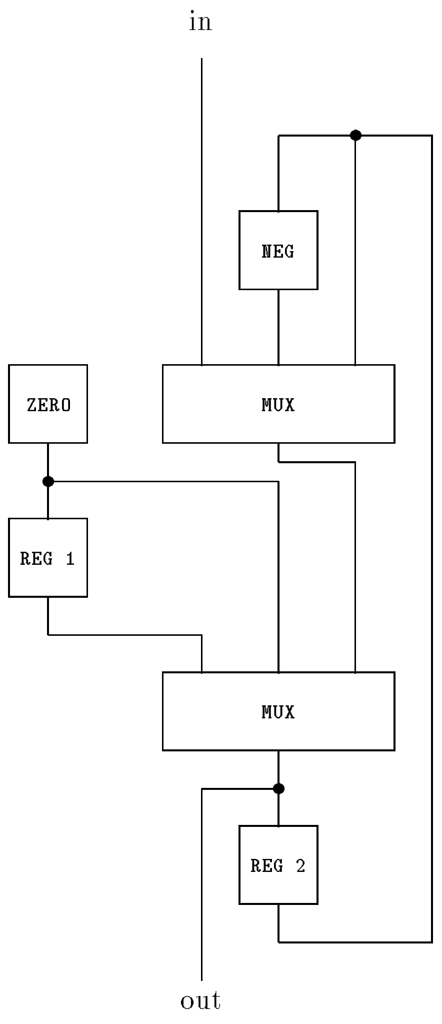
evenParity(inp) {all_binary(inp) & inp != Nil} $\rightarrow$ output(eParCirc(inp))

The result of replaying Figures 8.24 and 8.25 is given in Figures 8.27 and 8.28. These derivations show that replay is only partly successful; the case marked by a box needs further work as shown by the error messages in Figure 8.29. To complete the derivation, we reinvoke replay with the node marked by $\boxed{1}$ in Figure 8.24. This gives the derivation in Figure 8.30, which ReFocus constructs with no error messages.

[9]The `reorient` operator appearing in some of these cases is used to orient a property before expanding it.



a. Odd parity circuit.



b. Even parity circuit.

Figure 8.22: Implementations of oddParity and evenParity.

Discussion

ReFocus failed when constructing the derivations in Figures 8.27 and 8.28 because generalization matched the framed case to the sibling of node $\boxed{1}$ (marked by $\boxed{2}$ in Figure 8.25) rather than to $\boxed{1}$. The initial states of both cases (which are not shown) differ only in the placement of 0's and 1's, with the result that generalization is not able to adequately distinguish between them. Generalization also causes problems when completing the proof in Figure 8.30 and so was turned off. These problems indicate that further work is needed on applying generalization to ReFocus. However, generalization is useful for matching cases when constructing the proof in Figures 8.27 and 8.28.[10]

In any case, ReFocus does significantly reduce the amount of work for the user by replaying as much of the proof of oParCirc as possible. Even if the user does not recognize that the wrong case was matched during replay, replaying the proof of oParCirc constructs a large portion of the proof for eParCirc and the remainder can be constructed by hand. This illustrates that replay is useful even when it is successful on only part of a problem.

[10] Furthermore, turning generalization off does not improve the results of replay in 8.27 and 8.28.

precedence relations:
```
    oParCirc > opar > {all_binary, one, neg}
    oParCirc > initial_state
    neg > mux > {binary, output}
    oddParity > {output, oParCirc}
```
lexicographic opar(2, 1), mux(1, 2, 3)

precondition opar(s, l) : l != Nil & all_binary(l)
precondition precondition mux(a, b, c) : binary(a) & binary(b) & binary(c)

definitions:
```
    one() → 1
    mux(1, in1, in2) → in1
    mux(0, in1, in2) → in2

    oParCirc(input) → opar(initial_state(), 0.input)
    initial_state() → State(0,0,1)
    opar(State(r1,r2,out), i.Nil) → State(one(), out, mux(r1, mux(i, neg(out), out), one()))
    opar(State(r1,r2,out), i.input) input != Nil →
                opar(State(one(), out, mux(r1, mux(i, neg(out), out), one())), input)
    output(State(r1, r2, out)) → out
```

properties:
```
    neg(i) → mux(i, 0, 1)
    mux(i, 1, 0) → i
    neg(mux(i, a, b)) → mux(i, neg(a), neg(b))
```

Figure 8.23: Odd parity circuit specification.

```
Prove: oddParity(inp) {all_binary(inp) & inp != Nil} → output(oParCirc(inp))
script: simplify, rewrite, expand(oddParity(inp))

  > cases from oddParity(inp)

    > case inp == [0]:  *identity
    script: simplify, rewrite

    > case inp == [1]:  *identity
    script: simplify, rewrite

    > case inp == 1.rest & rest != Nil
    mux(output(opar(State(1, 1, 1), rest)), 0, 1) {rest != Nil & all_binary(rest)} →
        output(opar(State(1, 1, 0), rest))
    script: simplify, rewrite, expand(opar(State(1, 1, 1), rest))

      > cases from opar(State(1, 1, 1), rest)

        > case rest == [i]:  *identity
        script: simplify, rewrite

        > case rest == i.input & input != Nil
        output(opar(State(1, 1, 0), i.input))
            {(i == 0 & input != Nil & all_binary(input) |
            i == 1 & input != Nil & all_binary(input))} →
            mux(output(opar(State(1, 1, mux(i, 0, 1)), input)), 0, 1)
        script: simplify, reorient, expand(opar(State(1, 1, 0), i.input))

          > cases from opar(State(1, 1, 0), i.input)

            > case input == Nil:  *identity
            script: simplify, rewrite

            > case input != Nil
            mux(output(opar(State(1, 1, mux(i, 0, 1)), input)), 0, 1)
                {(i == 0 & input != Nil & all_binary(input) |
                i == 1 & input != Nil & all_binary(input))} →
                output(opar(State(1, 0, i), input))
            script: simplify, rewrite, reorient, expand(mux)

              > cases from mux(i, 0, 1)

                > case i == 1
                mux(output(opar(State(1, 1, 0), input)), 0, 1)
                    {input != Nil & all_binary(input)} =                    [1]
                    output(opar(State(1, 0, 1), input))
                script: simplify, rewrite, expand, add-inductive-hypotheses

                  > cases from opar(State(1, 1, 0), input)

                    > case input == [i]:  *identity
                    script: simplify, rewrite

                    > case input == i.input1 & input1 != Nil
                    output(opar(State(1, 0, 1), i.input1))
                        {(i == 0 & input1 != Nil & all_binary(input1) |
                        i == 1 & input1 != Nil & all_binary(input1))} →
                        mux(output(opar(State(1, 0, i), input1)), 0, 1)
                    script: simplify, rewrite, reorient, expand

                      > cases from opar(State(1, 0, 1), i.input1)
```

continued in the next figure ...

Figure 8.24: Correctness proof for oParCirc, part 1.

... continued from the previous figure

```
            > cases from opar(State(1, 0, 1), i.input1)
              > case input1 == Nil:  *identity
              script:  simplify, rewrite
              > case input1 != Nil
              output(opar(State(1, 1, mux(i, 0, 1)), input1))
                  {(i == 0 & input1 != Nil & all_binary(input1) |
                  i == 1 & input1 != Nil & all_binary(input1))} →
                  mux(output(opar(State(1, 0, i), input1)), 0, 1)
              script:  simplify, rewrite, reorient, expand(mux)
                > cases from mux(i, 0, 1)
                  > case i == 1:  *identity
                  script:  simplify, rewrite, rewrite(output), rewrite
                  > case i == 0
                  output(opar(State(1, 1, 1), input1))
                      {all_binary(input1) & input1 != Nil} →
                      mux(output(opar(State(1, 0, 0), input1)), 0, 1)
                  script:  simplify, rewrite, reorient, expand
                    > cases from opar(State(1, 1, 1), input1)
                      > case input1 == [i]:  *identity
                      script:  simplify, rewrite
                      > case input1 == i.input & input != Nil
                      mux(output(opar(State(1, 0, 0), i.input)), 0, 1)
                          {(i == 0 & input != Nil & all_binary(input) |
                          i == 1 & input != Nil & all_binary(input))} →
                          output(opar(State(1, 1, mux(i, 0, 1)), input))
                      script:  simplify, rewrite, reorient, expand
                        > cases from opar(State(1, 0, 0), i.input)
                          > case input == Nil:  *identity
                          script:  simplify, rewrite
                          > case input != Nil
                          output(opar(State(1, 1, mux(i, 0, 1)), input))
                              {(i == 0 & input != Nil & all_binary(input) |
                              i == 1 & input != Nil & all_binary(input))} →
                              mux(output(opar(State(1, 0, i), input)), 0, 1)
                          script:  simplify, rewrite, reorient, expand(mux)
                            > cases from mux(i, 0, 1)
                              > case i == 1:  *identity
                              script:  simplify, rewrite, rewrite(output), rewrite
                              > case i == 0:  *identity
                              script:  simplify, rewrite
        > case i == 0:  *identity                                                       [2]
        script:  simplify, rewrite
> case inp == 0.rest & rest != Nil:  *identity
script:  simplify, rewrite
```

Figure 8.25: Correctness proof for oParCirc, part 2.

precedence relations:
  eParCirc > epar > {all_binary, zero, neg}
  eParCirc > eInitial_state
  neg > mux > {binary, output}
  evenParity > {output, eParCirc}
lexicographic epar(2, 1), mux(1, 2, 3)

precondition epar(s, l) : l != Nil & all_binary(l)
definitions:
  zero() → 0
  eParCirc(input) → epar(eInitial_state(), 0.input)
  eInitial_state() → State(0,0, $\boxed{0}$)
  epar(State(r1,r2,out), i.Nil) →
    State($\boxed{\text{zero()}}$, out, mux(r1, $\boxed{\text{zero()}}$, $\boxed{\text{mux(i, neg(out) , out)}}$))
  epar(State(r1,r2,out), i.input) {input != Nil} →
    epar(State($\boxed{\text{zero()}}$, out, mux(r1, $\boxed{\text{zero()}}$, $\boxed{\text{mux(i, neg(out), out)}}$)), input)

Figure 8.26: Even parity circuit specification.

```
Prove:  evenParity(inp) {all_binary(inp) & inp != Nil} → output(eParCirc(inp))
script:  simplify, rewrite, expand(evenParity(inp))

  > cases from evenParity(inp)

    > case inp == [0]:  *identity
    script:  simplify, rewrite

    > case inp == [1]:  *identity
    script:  simplify, rewrite

    > case inp == 0.rest & rest != Nil:  *identity
    script:  simplify, rewrite

    > case inp == 1.rest & rest != Nil
    mux(output(epar(State(0, 0, 0), rest)), 0, 1) {rest != Nil & all_binary(rest)} →
        output(epar(State(0, 0, 1), rest))
    script:  simplify, rewrite, expand

      > cases from epar(State(0, 0, 0), rest)

        > case rest == [i]:  *identity
        script:  simplify, rewrite

        > case rest == i.input & input != Nil
        output(epar(State(0, 0, 1), i.input))
            {(i == 0 & input != Nil & all_binary(input) |
            i == 1 & input != Nil & all_binary(input))} →
            mux(output(epar(State(0, 0, mux(i, 1, 0)), input)), 0, 1)
        script:  simplify, reorient, expand

          > cases from epar(State(0, 0, 1), i.input)

            > case input == Nil:  *identity
            script:  simplify, rewrite

            > case input != Nil
            output(epar(State(0, 1, mux(i, 0, 1)), input))
                {(i == 0 & input != Nil & all_binary(input) |
                i == 1 & input != Nil & all_binary(input))} →
                mux(output(epar(State(0, 0, i), input)), 0, 1)
            script:  simplify, rewrite, reorient, expand(mux)

              > cases from mux(i, 0, 1)

                > case i == 1
                output(epar(State(0, 1, 0), input)) {input != Nil & all_binary(input)} =
                    mux(output(epar(State(0, 0, 1), input)), 0, 1)
                script:  simplify, rewrite

                > case i == 0
                output(epar(State(0, 1, 1), input)) {input != Nil & all_binary(input)} →
                    output(epar(State(0, 0, 1), input))
                script:  simplify, rewrite, expand

                  > cases from epar(State(0, 1, 1), input)

        continued in the next figure ...
```

Figure 8.27: Result of replaying Figure 8.25 to verify eParCirc, part 1.

. . . continued from the previous figure

```
        > cases from epar(State(0, 1, 1), input)

            > case input == [i]:  *identity
            script:  simplify, rewrite

            > case input == i.input1 & input1 != Nil
            output(epar(State(0, 0, 1), i.input1))
                {(i == 0 & input1 != Nil & all_binary(input1) |
                i == 1 & input1 != Nil & all_binary(input1))} →
                output(epar(State(0, 1, mux(i, 0, 1)), input1))
            script:  simplify, rewrite, reorient, expand

              > cases from epar(State(0, 0, 1), i.input1)

                > case input1 == Nil:  *identity
                script:  simplify, rewrite

                > case input1 != Nil:  *identity
                script:  simplify, rewrite

Replay... failed
```

Figure 8.28: Result of replaying Figure 8.25 to verify eParCirc, part 2.

```
Replay error #1 @ /7,1,4,1,2,1,2,1,1 ? case i == 1:
    Both initial and final states do not match prototype
    new: output(epar(State(0, 1, 0), input))
         {input != Nil & all_binary(input)} =
         mux(output(epar(State(0, 0, 1), input)), 0, 1)
    old: *identity

Replay error #2 @
    /7 P? evenParity(inp) {all_binary(inp) & inp != Nil} → output(eParCirc(inp)):
    Closed failed
```

Figure 8.29: Error output from replaying Figure 8.25 to verify eParCirc.

```
> case i == 1
output(epar(State(0, 1, 0), input)) {input != Nil & all_binary(input)} →
    mux(output(epar(State(0, 0, 1), input)), 0, 1)
script:  simplify, rewrite, expand, add-inductive-hypotheses

  > cases from epar(State(0, 1, 0), input)

    > case input == [i]:  *identity    script:  simplify, rewrite

    > case input == i.input1 & input1 != Nil
    mux(output(epar(State(0, 0, 1), i.input1)), 0, 1)
        {(i == 0 & input1 != Nil & all_binary(input1) |
        i == 1 & input1 != Nil & all_binary(input1))} →
        output(epar(State(0, 0, i), input1))
    script:  simplify, rewrite, reorient, expand

      > cases from epar(State(0, 0, 1), i.input1)

        > case input1 == Nil:  *identity    script:  simplify, rewrite

        > case input1 != Nil
        mux(output(epar(State(0, 1, mux(i, 0, 1)), input1)), 0, 1)
            {(i == 0 & input1 != Nil & all_binary(input1) |
            i == 1 & input1 != Nil & all_binary(input1))} →
            output(epar(State(0, 0, i), input1))
        script:  simplify, rewrite, reorient, expand(mux)

          > cases from mux(i, 0, 1)

            > case i == 1:  *identity
            script:  simplify, rewrite, rewrite(output), rewrite

            > case i == 0
            mux(output(epar(State(0, 0, 1), input1)), 0, 1)
                {all_binary(input1) & input1 != Nil} →
                output(epar(State(0, 0, 0), input1))
            script:  simplify, rewrite, reorient, expand

              > cases from epar(State(0, 0, 1), input1)

                > case input1 == [i]:  *identity    script:  simplify, rewrite

                > case input1 == i.input & input != Nil
                output(epar(State(0, 0, 0), i.input))
                    {(i == 0 & input != Nil & all_binary(input) |
                    i == 1 & input != Nil & all_binary(input))} →
                    mux(output(epar(State(0, 1, mux(i, 0, 1)), input)), 0, 1)
                script:  simplify, rewrite, reorient, expand

                > cases from epar(State(0, 0, 0), i.input)

                  > case input == Nil:  *identity    script:  simplify, rewrite

                  > case input != Nil
                  mux(output(epar(State(0, 1, mux(i, 0, 1)), input)), 0, 1)
                      {(i == 0 & input != Nil & all_binary(input) |
                      i == 1 & input != Nil & all_binary(input))} →
                      output(epar(State(0, 0, i), input))
                  script:  simplify, rewrite, reorient, expand(mux(i, 0, 1))

                    > cases from mux(i, 0, 1)

                      > case i == 1:  *identity
                      script:  simplify, rewrite, rewrite(output), rewrite

                      > case i == 0:  *identity    script:  simplify, rewrite
Replay... apparently succeeded
```

Figure 8.30: Repair of correctness proof for eParCirc.

precedence relations:

```
scan > {fword, rword} > skip > scanskip
{fword, rword} > scanword
scanword = scanskip
```

definitions:

```
scan(Nil)      → Nil
scan(ch.s)     → fword(ch.s).scan(rword(ch.s))
fword(Nil)     → Nil
fword(A(c).s)  → c.fword(s)
fword(S.s)     → Nil
rword(Nil)     → Nil
rword(A(c).s)  → rword(s)
rword(S.s)     → skip(s)
skip(Nil)      → Nil
skip(A(c).s)   → A(c).s
skip(S.s)      → skip(s)
```

Figure 8.31: Text scanner specification.

8.5.2 Breaking a Document into Words

Figure 8.31 gives a function, scan, which breaks a list of characters (a "document") into words. This function, taken from [Red88a, Fea79] with modifications, assumes that words are sequences of alphanumeric characters A(n) separated by spaces S. For example,

scan([A(1), S, A(2), A(3)]) = [[1], [2,3]]

illustrates splitting a document into two words. The **scan** function calls **fword** to process the first word and **rword** to process the rest of the document after skipping the first word and any following spaces.

Because both **fword** and **rword** both process the first word, **scan** examines each character twice. A one-pass program can be derived from the specification

scanword(s) = fword(s).scan(rword(s))

Figure 8.32 gives the resulting derivation, where an auxiliary function has been introduced to combine scanning words and skipping leading spaces.

Figure 8.33 gives a modified version of scan (in which boxes mark the modifications) to fix two bugs. The first bug, identified in [Red88a], is that documents starting with spaces result in empty words:

scan([S, A(1), A(2), S]) = [Nil, [1, 2]]

```
Focus: scanword(s) = fword(s).scan(rword(s))
[closed with program:]
scanword(Nil) → [Nil]
scanword(A(c).s1) → LET v0.v1 = scanword(s1) IN (c.v0).v1
scanword(S.s1) → Nil.scanskip(s1)
script: simplify, rewrite, expand(fword(s)), close-subsidiary

  Focus: scanskip(s) = scan(skip(s))
  [closed with program:]
  scanskip(Nil) → Nil
  scanskip(A(c).s1) → LET v0.v1 = scanword(s1) IN (c.v0).v1
  scanskip(S.s1) → scanskip(s1)
  script: simplify, rewrite, expand(skip(s))

    > cases from skip(s)

      > case s == Nil
      scanskip(Nil) = scan(Nil)
      scanskip(Nil) → Nil
      script: simplify, rewrite

      > case s == A(c).s1
      scanskip(A(c).s1) = scan(A(c).s1)
      scanskip(A(c).s1) → LET v0.v1 = scanword(s1) IN (c.v0).v1
      script: simplify, rewrite

      > case s == S.s1
      scanskip(S.s1) = scan(skip(s1))
      scanskip(S.s1) → scanskip(s1)
      script: simplify, rewrite

  > cases from fword(s)

    > case s == Nil
    scanword(Nil) = Nil.scan(rword(Nil))
    scanword(Nil) → [Nil]
    script: simplify, rewrite

    > case s == A(c).s1
    scanword(A(c).s1) = (c.fword(s1)).scan(rword(A(c).s1))
    scanword(A(c).s1) → LET v0.v1 = scanword(s1) IN (c.v0).v1
    script: simplify, rewrite

    > case s == S.s1
    scanword(S.s1) = Nil.scan(rword(S.s1))
    scanword(S.s1) → Nil.scanskip(s1)
    script: simplify, rewrite
```

Figure 8.32: Original derivation of scanword.

precedence relations:

$\mathsf{scan} > \mathsf{scanrest} > \{\mathsf{fword}, \mathsf{rword}\} > \mathsf{skip} > \mathsf{scanskip}$
$\{\mathsf{fword}, \mathsf{rword}\} > \mathsf{scanword}$
$\mathsf{scanword} = \mathsf{scanskip}$

definitions:

$$
\begin{array}{l}
\boxed{\begin{array}{lll}
\mathsf{scan(s)} & \rightarrow & \mathsf{scanrest(skip(s))}\\
\mathsf{scanrest(Nil)} & \rightarrow & \mathsf{Nil}\\
\mathsf{scanrest(ch.s)} & \rightarrow & \mathsf{fword(ch.s).scanrest(rword(ch.s))}
\end{array}}\\
\begin{array}{lll}
\mathsf{fword(Nil)} & \rightarrow & \mathsf{Nil}\\
\mathsf{fword(A(c).s)} & \rightarrow & \boxed{\mathsf{A(c)}}\mathsf{.fword(s)}\\
\mathsf{fword(S.s)} & \rightarrow & \mathsf{Nil}
\end{array}\\
\boxed{\begin{array}{lll}
\mathsf{fword(L.s)} & \rightarrow & \mathsf{Nil}
\end{array}}\\
\begin{array}{lll}
\mathsf{rword(Nil)} & \rightarrow & \mathsf{Nil}\\
\mathsf{rword(A(c).s)} & \rightarrow & \mathsf{rword(s)}\\
\mathsf{rword(S.s)} & \rightarrow & \mathsf{skip(s)}
\end{array}\\
\boxed{\begin{array}{lll}
\mathsf{rword(L.s)} & \rightarrow & \mathsf{skip(s)}
\end{array}}\\
\begin{array}{lll}
\mathsf{skip(Nil)} & \rightarrow & \mathsf{Nil}\\
\mathsf{skip(A(c).s)} & \rightarrow & \mathsf{A(c).s}\\
\mathsf{skip(S.s)} & \rightarrow & \mathsf{skip(s)}
\end{array}\\
\boxed{\begin{array}{lll}
\mathsf{skip(L.s)} & \rightarrow & \mathsf{skip(s)}
\end{array}}
\end{array}
$$

Figure 8.33: Modified scan specification.

The modifications fix this by skipping leading spaces before scanning for words. The second bug is that the specification violates the general principle that output should be legal input: fword converts $\mathsf{A}(n)$ to the non-character symbol n. These are also fixed by the modifications. In addition, support has been added to treat linefeeds L as spaces.

Replaying the derivation in Figure 8.32 gives Figure 8.34. Comparing the old and new specifications of scan leads ReFocus to update the specification of scanword to call scanrest rather than scan. Because the new program (like the original) does not contain any precedence violations, ReFocus accepts it in spite of a number of differences. Thus ReFocus constructs the new derivation without any error messages. This illustrates using replay to update derivations after fixing specifications.

We next consider adding support for a command language in the style of the Unix `nroff` text processing system. Assume such commands are denoted by strings starting with a special marker C and ending with L.[11] Figure 8.35 defines a version of scan supporting such commands.

[11]To process `nroff` documents, we would modify the input reader to substitute `C` for periods that begin a line.

```
Focus: scanword(s) = fword(s).scanrest(rword(s))
[closed with program:]
scanword(Nil) → [Nil]
scanword(A(c).s1) → LET v0.v1 = scanword(s1) IN (A(c).v0).v1
scanword(S.s1) → Nil.scanskip(s1)
scanword(L.s1) → Nil.scanskip(s1)
script: simplify, rewrite, expand(fword(s)), close-subsidiary

  Focus: scanskip(s) = scanrest(skip(s))
  [closed with program:]
  scanskip(Nil) → Nil
  scanskip(A(c).s1) → LET v0.v1 = scanword(s1) IN (A(c).v0).v1
  scanskip(S.s1) → scanskip(s1)
  scanskip(L.s1) → scanskip(s1)
  script: simplify, rewrite, expand(skip(s))

    > cases from skip(s)

      > case s == Nil
      scanskip(Nil) → Nil
      script: simplify, rewrite

      > case s == A(c).s1
      scanskip(A(c).s1) → LET v0.v1 = scanword(s1) IN (A(c).v0).v1
      script: simplify, rewrite

      > case s == S.s1
      scanskip(S.s1) → scanskip(s1)
      script: simplify, rewrite

      > case s == L.s1
      scanskip(L.s1) → scanskip(s1)
      script: simplify, rewrite

  > cases from fword(s)

    > case s == Nil
    scanword(Nil) → [Nil]
    script: simplify, rewrite

    > case s == A(c).s1
    scanword(A(c).s1) → LET v0.v1 = scanword(s1) IN (A(c).v0).v1
    script: simplify, rewrite

    > case s == S.s1
    scanword(S.s1) → Nil.scanskip(s1)
    script: simplify, rewrite

    > case s == L.s1
    scanword(L.s1) → Nil.scanskip(s1)
    script: simplify, rewrite

Replay... apparently succeeded
```

Figure 8.34: Result of replaying Figure 8.32 after first set of modifications to scan.

This version introduces a number of functions (cmd, word, skipcmd, and skipword) to separate scanning words from scanning commands.

Because the new version of scan is very different from that in Figure 8.33, it is not clear that replay will be successful. Figure 8.36 shows that it is not: since the expressions marked by boxes do not call scanword and scanskip, the resulting program still makes two passes over the input. To derive a one-pass program for the specification in Figure 8.35, the user must introduce more auxiliary functions to combine scanning and skipping for both commands and words. Thus replay fails in this case because it *should* fail; despite similarities, the new problem is not very close to the original.

Discussion

This example illustrates that the success of replay depends on problems having similar solutions, not on the problems themselves being similar. It also illustrates the importance of automatically checking the results of derivations. Given the complexity of the program in Figure 8.36, it would be easy to miss that replay failed. However, ReFocus notes that the new program contains precedence violations and so warns that it may not be acceptable by producing the error messages in Figure 8.37. Thus while replay fails, ReFocus does not.

As described earlier in this section, one way to repair the derivation is to introduce more auxiliary functions. However, this makes the derivation complicated because each auxiliary function must call the others. An alternative approach is to rewrite the specification. Figure 8.38 defines a version of scan in which state variables are used to describe the object, either a word or a command, being processed at each moment. The valid states are None, Command, and Word. Because the structure of this program is closer to the version of scan in Figure 8.33, replaying the derivation in Figure 8.34 is successful and gives the derivation in Figures 8.39 and 8.40. Since the new program contains no precedence violations, ReFocus accepts it without any error messages.

precedence relations:

```
scan > scanrest > {fword, rword} > skip > scanskip
fword > {cmd, word}
rword > {skipcmd, skipword} > skip
scanword = scanskip
```

definitions:

```
scan(s)            → scanrest(skip(s))
scanrest(Nil)      → Nil
scanrest(c.s)      → fword(c.s).scanrest(rword(c.s))

fword(C.s)         → C.cmd(s)
fword(A(c).s)      → A(c).word(s)

cmd(Nil)           → Nil
cmd(A(c).s)        → A(c).cmd(s)
cmd(S.s)           → S.cmd(s)
cmd(L.s)           → Nil

word(Nil)          → Nil
word(C.s)          → Nil
word(A(c).s)       → A(c).word(s)
word(S.s)          → Nil
word(L.s)          → Nil

rword(C.s)         → skipcmd(s)
rword(A(c).s)      → skipword(s)

skipcmd(Nil)       → Nil
skipcmd(L.s)       → skip(s)
skipcmd(A(c).s)    → skipcmd(s)
skipcmd(S.s)       → skipcmd(s)

skipword(Nil)      → Nil
skipword(C.s)      → C.s
skipword(A(c).s)   → skipword(s)
skipword(S.s)      → skip(s)
skipword(L.s)      → skip(s)

skip(Nil)          → Nil
skip(S.s)          → skip(s)
skip(L.s)          → skip(s)
skip(C.s)          → C.s
skip(A(c).s)       → A(c).s
```

Figure 8.35: Initial version of scan supporting commands.

Focus: scanword(s) = fword(s).scanrest(rword(s))
[*closed with program:*]
scanword(C.s1) → (C.cmd(s1)).scanrest(skipcmd(s1))
scanword(A(c).s1) → (A(c).word(s1)).scanrest(skipword(s1))
script: `simplify, rewrite, expand(fword(s)), close-subsidiary`

Focus: scanskip(s) = scanrest(skip(s))
[*closed with program:*]
scanskip(Nil) → Nil
scanskip(S.s1) → scanskip(s1)
scanskip(L.s1) → scanskip(s1)
scanskip(C.s1) → (C.cmd(s1)).scanrest(skipcmd(s1))
scanskip(A(c).s1) → (A(c).word(s1)).scanrest(skipword(s1))
script: `simplify, rewrite, expand(skip(s))`

> *cases from* skip(s)

> *case* s == Nil
scanskip(Nil) = scanrest(Nil)
scanskip(Nil) → Nil
script: `simplify, rewrite`

> *case* s == S.s1
scanskip(S.s1) = scanrest(skip(s1))
scanskip(S.s1) → scanskip(s1)
script: `simplify, rewrite`

> *case* s == L.s1
scanskip(L.s1) = scanrest(skip(s1))
scanskip(L.s1) → scanskip(s1)
script: `simplify, rewrite`

> *case* s == C.s1
scanskip(C.s1) = scanrest(C.s1)
$\boxed{\text{scanskip(C.s1) = (C.cmd(s1)).scanrest(skipcmd(s1))}}$
script: `simplify, rewrite`

> *case* s == A(c).s1
scanskip(A(c).s1) = scanrest(A(c).s1)
$\boxed{\text{scanskip(A(c).s1) = (A(c).word(s1)).scanrest(skipword(s1))}}$
script: `simplify, rewrite`

> *cases from* fword(s)

> *case* s == C.s1
scanword(C.s1) = (C.cmd(s1)).scanrest(rword(C.s1))
$\boxed{\text{scanword(C.s1) = (C.cmd(s1)).scanrest(skipcmd(s1))}}$
script: `simplify, rewrite`

> *case* s == A(c).s1
scanword(A(c).s1) = (A(c).word(s1)).scanrest(rword(A(c).s1))
$\boxed{\text{scanword(A(c).s1) = (A(c).word(s1)).scanrest(skipword(s1))}}$
script: `simplify, rewrite`

`Replay... failed`

Figure 8.36: Result of replaying Figure 8.34 after first attempt at introducing commands.

```
Replay error #1 @ /4,3,1,1,4 ? case s == C.s1:
    Both initial and final states do not match prototype
    new: scanskip(C.s1) = (C.cmd(s1)).scanrest(skipcmd(s1))
         violations: {scanskip ⊁ cmd, scanskip ⊁ skipcmd, scanskip ⊁ scanrest}
    old: scanskip(A(c).s1) → LET v0.v1 = scanword(s1) IN (A(c).v0).v1
         violations: none

Replay error #2 @ /4,3,1,1,5 ? case s == A(c).s1:
    Both initial and final states do not match prototype
    new: scanskip(A(c).s1) = (A(c).word(s1)).scanrest(skipword(s1))
         violations: {scanskip ⊁ word, scanskip ⊁ skipword, scanskip ⊁ scanrest}
    old: scanskip(A(c).s1) → LET v0.v1 = scanword(s1) IN (A(c).v0).v1
         violations: none

Replay error #3 @ /4,3,2,1 ? case s == C.s1:
    Both initial and final states do not match prototype
    new: scanword(C.s1) = (C.cmd(s1)).scanrest(skipcmd(s1))
         violations: {scanword ⊁ cmd, scanword ⊁ skipcmd, scanword ⊁ scanrest}
    old: scanword(A(c).s1) → LET v0.v1 = scanword(s1) IN (A(c).v0).v1
         violations: none

Replay error #4 @ /4,3,2,2 ? case s == A(c).s1:
    Both initial and final states do not match prototype
    new: scanword(A(c).s1) = (A(c).word(s1)).scanrest(skipword(s1))
         violations: {scanword ⊁ word, scanword ⊁ skipword, scanword ⊁ scanrest}
    old: scanword(A(c).s1) → LET v0.v1 = scanword(s1) IN (A(c).v0).v1
         violations: none
```

Figure 8.37: Error messages for the failed replay in Figure 8.36.

precedence relations:

```
    scan > scanrest > {fword, rword} > skip > scanskip
    {fword, rword} > scanword
    scanword = scanskip
```

definitions:

```
    scan(s)                      → scanrest(skip(s))
    scanrest(Nil)                → Nil
    scanrest(c.s)                → fword(c.s, None).scanrest(rword(c.s, None))

    fword(C.s,    None)          → C.fword(s, Command)
    fword(A(c).s, None)          → A(c).fword(s, Word)

    fword(Nil,    Command)       → Nil
    fword(A(c).s, Command)       → A(c).fword(s, Command)
    fword(S.s,    Command)       → S.fword(s, Command)
    fword(L.s,    Command)       → Nil

    fword(Nil,    Word)          → Nil
    fword(C.s,    Word)          → Nil
    fword(A(c).s, Word)          → A(c).fword(s, Word)
    fword(S.s,    Word)          → Nil
    fword(L.s,    Word)          → Nil

    rword(C.s,    None)          → rword(s, Command)
    rword(A(c).s, None)          → rword(s, Word)

    rword(Nil,    Command)       → Nil
    rword(L.s,    Command)       → skip(s)
    rword(A(c).s, Command)       → rword(s, Command)
    rword(S.s,    Command)       → rword(s, Command)

    rword(Nil,    Word)          → Nil
    rword(C.s,    Word)          → C.s
    rword(A(c).s, Word)          → rword(s, Word)
    rword(S.s,    Word)          → skip(s)
    rword(L.s,    Word)          → skip(s)

    skip(Nil)                    → Nil
    skip(S.s)                    → skip(s)
    skip(L.s)                    → skip(s)
    skip(C.s)                    → C.s
    skip(A(c).s)                 → A(c).s
```

Figure 8.38: Revised scan specification supporting commands.

```
Focus: scanword(s, state) = fword(s, state).scanrest(rword(s, state))
[closed with program:]
scanword(C.s1,    None)      → LET v0.v1 = scanword(s1, Command) IN (C.v0).v1
scanword(A(c).s1, None)      → LET v0.v1 = scanword(s1, Word) IN (A(c).v0).v1
scanword(Nil,     Command)   → [Nil]
scanword(A(c).s1, Command)   → LET v0.v1 = scanword(s1, Command) IN (A(c).v0).v1
scanword(S.s1,    Command)   → LET v0.v1 = scanword(s1, Command) IN (S.v0).v1
scanword(L.s1,    Command)   → Nil.scanskip(s1)
scanword(Nil,     Word)      → [Nil]
scanword(C.s1,    Word)      → LET v0.v1 = scanword(s1, Command) IN Nil.(C.v0).v1
scanword(A(c).s1, Word)      → LET v0.v1 = scanword(s1, Word) IN (A(c).v0).v1
scanword(S.s1,    Word)      → Nil.scanskip(s1)
scanword(L.s1,    Word)      → Nil.scanskip(s1)
script: simplify, rewrite, expand(fword(s)), close-subsidiary

  Focus: scanskip(s) = scanrest(skip(s))
  [closed with program:]
  scanskip(Nil)       → Nil
  scanskip(S.s1)      → scanskip(s1)
  scanskip(L.s1)      → scanskip(s1)
  scanskip(C.s1)      → LET v0.v1 = scanword(s1, Command) IN (C.v0).v1
  scanskip(A(c).s1)   → LET v0.v1 = scanword(s1, Word) IN (A(c).v0).v1
  script: simplify, rewrite, expand(skip(s))

   > cases from skip(s)

    > case s == Nil: scanskip(Nil) → Nil
    script: simplify, rewrite

    > case s == S.s1
    scanskip(S.s1) → scanskip(s1)
    script: simplify, rewrite

    > case s == L.s1
    scanskip(L.s1) → scanskip(s1)
    script: simplify, rewrite

    > case s == C.s1
    scanskip(C.s1) → LET v0.v1 = scanword(s1, Command) IN (C.v0).v1
    script: simplify, rewrite

    > case s == A(c).s1
    scanskip(A(c).s1) → LET v0.v1 = scanword(s1, Word) IN (A(c).v0).v1
    script: simplify, rewrite

  > cases from fword(s, state)

continued in the next figure ...
```

Figure 8.39: Result of replaying Figure 8.34 after introducing states, part 1.

...continued from the previous figure

> *cases from* fword(s, state)

> *case* state == None & s == C.s1
scanword(C.s1, None) → LET v0.v1 = scanword(s1, Command) IN (C.v0).v1
script: `simplify, rewrite`

> *case* state == None & s == A(c).s1
scanword(A(c).s1, None) → LET v0.v1 = scanword(s1, Word) IN (A(c).v0).v1
script: `simplify, rewrite`

> *case* state == Command & s == Nil
scanword(Nil, Command) → [Nil]
script: `simplify, rewrite`

> *case* state == Command & s == A(c).s1
scanword(A(c).s1, Command) → LET v0.v1 = scanword(s1, Command) IN (A(c).v0).v1
script: `simplify, rewrite`

> *case* state == Command & s == S.s1
scanword(S.s1, Command) → LET v0.v1 = scanword(s1, Command) IN (S.v0).v1
script: `simplify, rewrite`

> *case* state == Command & s == L.s1
scanword(L.s1, Command) → Nil.scanskip(s1)
script: `simplify, rewrite`

> *case* state == Word & s == Nil
scanword(Nil, Word) → [Nil]
script: `simplify, rewrite`

> *case* state == Word & s == C.s1
scanword(C.s1, Word) → LET v0.v1 = scanword(s1, Command) IN Nil.(C.v0).v1
script: `simplify, rewrite`

> *case* state == Word & s == A(c).s1
scanword(A(c).s1, Word) → LET v0.v1 = scanword(s1, Word) IN (A(c).v0).v1
script: `simplify, rewrite`

> *case* state == Word & s == S.s1
scanword(S.s1, Word) → Nil.scanskip(s1)
script: `simplify, rewrite`

> *case* state == Word & s == L.s1
scanword(L.s1, Word) → Nil.scanskip(s1)
script: `simplify, rewrite`

```
Replay... apparently succeeded
```

Figure 8.40: Result of replaying Figure 8.34 after introducing states, part 2.

8.6 Conclusion

The main purpose of this chapter has been to show the results of applying ReFocus on a number of examples. These include the examples from the preceding chapters, examples from published papers, and moderately-sized examples. This establishes that ReFocus can solve the issues raised in the preceding chapters. At the same time, we have explored some of the limitations of ReFocus and identified ways to improve robustness.

A secondary purpose has been to demonstrate the usefulness of replay. While it may seem obvious that replay is a useful tool, an implementation is needed before conclusions can be drawn. ReFocus is only a prototype, but it does illustrate that replay is useful on (at least) moderately-sized problems. In particular, both the circuit-validation and text scanner examples show that replay can significantly reduce the user's workload.

While the examples show that replay is useful, they are a long way from demonstrating that the transformational implementation model is superior to traditional program development methods. In our experiences with Focus, it takes much longer to derive a program than to write it by hand, so it is not yet cost-effective to use such systems to maintain software. Transformational implementation is relatively new, and much work is needed on theory, tools and methods before it becomes a practical way to develop systems. Analyzing the usefulness of replay as a program maintenance tool must wait until more mature transformational implementation systems are available.

Though this work does not prove that replay simplifies maintaining real programs, it does succeed in showing that replay is useful. All of the present researchers in the Focus group use replay to help develop derivations. Though specifications may be easier to modify and debug than implementations, they are not very easy to write. It is rare that an initial attempt is complete and free of errors. Constructing derivations often reveals problems such as missing preconditions or omitted cases. Replay allows the user to reconstruct a derivation after modifying the specification so that discovering an error does not mean starting over. Thus this research shows that replay is useful even in today's systems.

Chapter 9

Conclusion

The transformational implementation model is a promising, though radical, solution to the problem of increasing the amount of automation in software development. In this paradigm, the user writes a formal specification and then applies equivalence-preserving rules to derive an efficient implementation. One benefit is increased consistency. Assuming the rules and inference engine are sound, the derived implementation will be consistent with its specification. A second benefit is simplified maintenance. Because there is a formal link between a specification and its implementation, the user can make changes to the specification and replay the derivation to recreate the implementation. A third benefit is design reuse. Since problems often share an underlying structure, replay can be used to apply the solution from one problem to another. This research confirms the usefulness of replay and proposes solutions to some of the problems which arise.

Program development systems—or more generally, proof development systems—can be classified by how search is controlled. In automated systems, such as [Bha91, Car86, HA88, KW94, Sil86, Vel92], the system controls search. The task of the user is to specify what problem is to be solved. In interactive systems, such as [Con86, Gor88, Pau86, Red88a], search is controlled by the user. The task of the system is to ensure that only sound inferences are made.

These two types of systems use replay in different ways. In automated systems, replay is used for *speedup*: by reusing derivations, significantly less search is needed to solve new problems. Besides finding solutions more quickly, replay can often find solutions to large problems for which standard search would exceed resource limits. Most work on replay falls into this category [Bax92, Bha91, Car86, HA88, KW94, Sil86, Vel92]. In interactive systems, on the other hand, replay is used to *increase automation*: by reusing derivations constructed by the user, the system is able to find solutions automatically even though it has no search control mechanism of its own. The replay mechanism described in this thesis falls into this second category. In contrast to replay in automated systems, little work has been done on replay in interactive systems.

Replay in interactive systems has a number of characteristics that differentiate it from replay in automated systems. One is the need for *flexibility.* When the user invokes replay, both the target and prototype are specified, implying that the two problems are similar. Thus replay must identify the similarities and apply the appropriate portions of the prototype derivation to the new problem. In contrast, replay in automated systems usually selects the prototype, so failing to identify similarities simply means that another prototype should be selected. Furthermore, if all else fails, the system can resort to search. Thus flexibility is a less important issue in automated systems.

A second difference is in *how replay is applied.* In interactive systems, replay can be used both to apply solutions to new problems and to repeat derivations after making modifications to specifications. In automated systems, replay is only useful for solving new problems. This means that replay in interactive systems is more likely to be applied when the differences between the problems are minor.

A third, and perhaps the most important difference, is the *scope* of the system. Research suggests that general problem solvers do not exist; *cf.* [RW88]. Thus automated systems must be built to solve problems in specific domains. For instance, a system may be designed to isolate variables in certain classes of equations [Sil86] or to construct programs in restricted domains [Bha91]. But because interactive systems rely on the user for search control, they can be more general. This means that replay must be applicable to an equally wide range of problems. It also means that goals are not clearly defined, making it more difficult to determine success or failure.

This thesis identifies several issues which must be addressed by replay in interactive systems. One issue is that the inference system should provide *large-grained operations.* Operations which refer to the details of the specification are too brittle and so likely to fail during replay. Large-grained operations promote flexibility. A second issue is *difference propagation.* Since the user must remain in control, operations cannot be so large-grained that they are not at all specific to the problem. Thus there are always references to the specification in the derivation history. To increase flexibility, replay must update the operations to refer to the new problem. A third issue is *acceptance testing.* Replay must determine success or failure so that the user does not need to check results manually. Because the goals of derivations in interactive systems are poorly defined, an important part of implementing replay is identifying ways to test results for acceptability. A final issue is *recovery from failure.* For flexibility, replay should attempt to reapply as much from the prototype's derivation history as possible before halting. This can range from simply switching between subproblems after failure to repairing derivations by identifying missing or unnecessary steps. Our experience in implementing replay suggests that these are the key problems of incorporating replay into interactive systems.

Because they are connected so closely to the underlying proof development system, the first and fourth issues (grain size and recovery from failure) do not permit general solutions. On the other hand, propagating differences and testing for acceptability are more general problems. The bulk of this thesis presents our solution to propagating differences. This is done by using second-order generalization to construct syntactic analogies. We give a formal definition of second-order generalization and show that it is computable. We then identify a useful subset of such generalizations and give a reasonably efficient algorithm to compute them.

The remainder of this thesis describes a heuristic for testing the acceptability of program derivations in transformational implementation systems. We show that term orderings can be used to measure progress by identifying the parts of equations which block them from being oriented as rewrite rules. This gives a test which is more robust than simply expecting exact matches.

The following sections discuss the contributions this thesis makes to unification theory, analogical reasoning, and formal program development.

9.1 Contributions to Unification Theory

Abstractly, unification is the process of solving equations by finding term substitutions for variables. Unification is important to many areas of computer science, including theorem proving [Rob65], programming languages [KK71, Kow79], and rewriting [DJ90, KB70]. The complementary problem, generalization, is the process of identifying commonalities between sets of terms (empirical induction) by introducing variables to abstract dissimilar subterms [Plo70, Rey70]. The main application of generalization is in machine learning [Fal88, FP90, Hal89, MCM83, MCM86, Owe90, Plo71], but generalization has also been applied to other problems such as type inference [NP92] and constructing programs [Bau79, Pla80]. This thesis—specifically, the work on second-order generalization—contributes to unification theory by extending it in a number of directions.

The main contribution of this thesis is the analysis of second-order generalization in Chapters 5 and 6. While higher-order unification has been defined for some time [Hue75, PJ72], higher-order generalization has not received much attention. The obvious approach, using the substitution ordering to define maximally instantiated terms, does not give useful results because second-order substitutions are too flexible and so too many terms are instances of each other. Instead, we base generalization on uniquely minimal complete sets of generalizations as defined in Chapter 5. We then show that uniquely minimal complete sets of generalizations exist for terms over monadic function symbols (Theorem 5.16) but not for the obvious extension for polyadic functions (Theorem 5.34). However, we show that the sets do exist when terms are restricted so that substitution preserves data (Theorem 5.38). We also identify restrictions on the variables that can occur in generalizations and give a natural algorithm (5.46) for computing such generalizations. Finally, in Chapter 6 we identify a useful subset of generalizations based on maximizing their size and give a polynomial-time algorithm for computing this subset. Thus this thesis contributes to unification theory by analyzing many aspects of second-order generalization.

A second contribution to unification theory is the use of categorical combinators [Cur93] to represent second-order terms (Section 5.1.1). [Bel90] shows that using categorical combinators has the advantage of allowing second-order matching to be expressed as first-order matching modulo the theory of categorical products. Our work shows that the categorical notation has the advantage of providing a natural basis for restricting terms so that second-order generalizations exist. The connection between generalization and unification suggests that other forms of categorical combinators may lead to improvements in unification.

9.2 Contributions to Analogical Reasoning

Analogical reasoning is the process of reusing solutions (*cf.* [Pol57]). In his survey of computational approaches to analogy, Hall [Hal89] identifies four components of analogical reasoning:

- *recognition*: finding a candidate prototype;
- *elaboration*: constructing analogical maps from the prototype to the target;
- *evaluation*: justifying, extending, and refining the analogical maps; and
- *consolidation*: putting the analogical maps into a form that is applicable to other problems.

While a more thorough treatment of evaluation and consolidation is left as future work,[1] this thesis contributes to analogical reasoning by providing a tool—second-order generalization—for implementing the second component, elaboration.

We describe a novel approach to constructing analogical maps based on syntactic information. As described in Chapter 4, this follows the work of [Coo90a, Coo90b, Eva68, FFG89, Fal88, Owe90] and others in using syntactic information to constrain analogy. In Chapters 5 and 6, we show how to construct analogical maps by finding their second-order generalization and pairing the substitutions. We then show that because these maps abstract common subterms, they are useful for transferring information between program derivations. This provides

[1]Recognition is not an issue when replaying derivations in an interactive environment; the user is responsible for finding an appropriate prototype.

a very simple but flexible model of analogical reasoning based purely on syntactic information. Furthermore, because the model does not depend on the underlying proof development system, incorporating it into other systems should not require major effort.

While this thesis applies second-order generalization only to replaying derivations, we believe this technique is applicable to other problems as well. For example, it could be applied to identifying differences between versions of programs at a more abstract level than that of characters. This might provide help, say, in finding recently-introduced errors in programs by directing a reviewer's attention to significant changes in code [LS92]. Furthermore, second-order generalization's ability to find close matches should be applicable to several problems. One is detecting plagiarism on programming assignments [Gri81]; by comparing key functions in programs, second-order generalization can suggest which solutions may not be original. A second application is reusing code stored in a library [NTFT91]. Current transformational implementation systems are too weak to solve large problems, but formal methods can be introduced into software development almost immediately by creating libraries of specifications and implementations. The difficulty is that no two programmers are likely to specify a problem in precisely the same way. After writing a specification, a programmer could use second-order generalization to find the closest matches in the library and then incorporate the implementation in the system being developed.

9.3 Contributions to Formal Program Development

Balzer, Cheatham, and Green [BCG83] suggest that the ability to maintain specifications rather than implementations is one of the primary advantages of developing programs formally. Rather than modifying code, programmers—or possibly even end users in restricted domains—can modify specifications and use replay to recreate programs with minimal effort. This should be an improvement because specifications appear to be easier to modify, leading to introducing fewer errors during maintenance. While much has been accomplished, significant research is needed before this paradigm is realized. More powerful tools are needed to handle larger and more abstract specifications, support more efficient implementations, and provide more assistance to the developer. However, even though a full evaluation of the usefulness of replay must wait until more powerful systems are available, this thesis does show that building an effective replay mechanism is feasible.

The primary contribution of this thesis to formal program development is a practical demonstration of the replayability of program derivations. In Chapter 2, we outline the requirements of an effective replay system. In Chapter 3, we describe Focus and its replay mechanism, ReFocus. In Chapter 6, we give a method for using syntactic analogy to transfer information between derivations. In Chapter 8, we apply ReFocus to a number of examples. Thus in contrast to previous work on replay in interactive environments [Gol90, MB87, MF89, SM85, Ste87, Wil83], we give a complete, robust implementation of replay. This justifies one of the basic assumptions of [BCG83] and others: replaying derivations is both feasible and useful.

A second contribution to formal program development is the application of term orderings to acceptance testing. It is not sufficient to simply replay derivations; the system must also verify that the resulting program meets the user's goals. In Chapter 7, we give a heuristic for checking acceptability using term orderings. These are orderings used in rewrite-based systems, such as Focus, to ensure that rewriting terminates. By verifying that each rewrite rule is consistent with a well-founded ordering, we can show that any sequence of rewrites will terminate. This thesis shows that term orderings can also be used to capture information about goals. Because the difference between specifications and programs is that specifications are unorientable, derivations make progress by removing subterms which block orientation. Comparing sets of such terms, known as *precedence violations*, gives a metric for measuring progress during replay. Because term orderings are already used in the system, this test has the advantage that the user does not need to enter additional information. The alternative, requiring the user to specify the expected form of the resulting program [Wil83], imposes a significant burden on the user.

Finally, this research shows that an effective replay mechanism does not require a detailed analysis of derivations. In contrast, the methods described in [Car86, Vel92] are based on detailed information such as what rules were applied and which search paths were examined and rejected. Such extensive information provides a complete picture of constructing derivations, and theoretically should allow derivations to be replayed without failure. However, analyzing such large amounts of information is time-consuming, difficult to implement, and strongly dependent on the implementation of the underlying theorem prover. The implementation of replay described in Chapter 3, ReFocus, depends little on the details of the underlying system, Focus. Instead, ReFocus uses second-order generalization to compare the terms in the prototype and target derivations. Because these methods depend only on information in the scripts and specifications, they can also be applied to other program development—and proof development—systems. Adding application-specific knowledge can improve replay, but this thesis shows that even a simple replay mechanism improves the process of developing programs (or proofs) by allowing the user to backtrack and recreate them after improving the specification.

9.4 Future Work

Many researchers [BCG83, Dar81, Fic85, Red90a, SS83, Wil83] have suggested that replay provides the key to permitting reuse of designs rather than implementations. But before we can claim this thesis is proven, program-development systems need to be improved to handle much larger problems. Until such systems are built, the usefulness of replay as a general development tool for real applications cannot be confirmed. When, and if, such systems are built, this thesis needs to be tested more completely.

A second area for future work is to improve the construction and application of analogical maps. While this thesis demonstrates that syntactic analogies constructed from second-order generalizations can transform expressions successfully in small examples, a more complete treatment of analogical reasoning should improve robustness. In presenting an algorithm for constructing analogical maps based on second-order generalization, this thesis considers only the elaboration component of analogical reasoning. A complete treatment of analogical reasoning in replay must also consider evaluation and consolidation [Hal89]. The evaluation mechanism described in this thesis is simplistic: apply the analogical maps and let replay tell the user if it fails. Failures could be used to refine the analogical maps, type information could be used to ensure that transformed terms are well-formed, and transformed subterms could be refined to ensure they match some part of the new term. The consolidation mechanism is even more simplistic: analogical maps are simply stored and reused without any concern for interactions with other maps. The system should use historical information to ensure that new analogical maps are either consistent with or orthogonal to old ones. These issues are not considered in this thesis to avoid obscuring the usefulness of second-order generalization, but a more complete treatment of analogical reasoning would be important to improving replay's robustness.

A third area for future work is to apply second-order generalization to other problems. We have suggested that it might be useful as a program difference tool, but there are probably other domains with rich term structures for which second-order generalization would be useful.

A fourth area is to investigate the connection between linear logic and unification. This may lead to identifying new classes of terms for which higher-order unification is decidable. Work such as [Far88, Mil91, MN87, Pre94] identify a number of useful classes, but most of these are based on higher-order patterns. Since patterns are very restricted, it is likely that other classes will give useful results.

Finally, a fifth area is to extend generalization in three directions. First, the restrictions on maximally specific condensed generalizations should be refined. The restriction against adjacent variables should be loosened to allow terms from distinct domains to have useful generalizations without using the $@_n$ notation. Also, the restriction against functions returning pair types should be relaxed. Analysis of the proof for Theorem 5.38 in Section B.2 suggests that this restriction is stronger than necessary. Second, the definition of second-order generalization should be extended to generalize types as well as terms. This can probably be done by applying the methods of [Pfe91]. Types provide powerful restrictions, and generalizing types may help the number of ambiguous or incorrect transform rules. They can also help by constraining which symbols are matched when generalizing terms from distinct domains. Third, generalization should be extended to higher orders. This would allow it to be applied to programming languages with higher-order type systems. It would also allow generalization to capture such concepts as switching the order in which functions are applied. Since matching terms apparently plays a key role in generalization, the fact that third and fourth-order matching have been found to be decidable [Dow92, Hue94] suggests that properly-defined higher-order generalization may be decidable as well. Generalizing types and higher-order terms will likely lead to even more powerful tools for syntactic analogies in domains with rich term structure.

Appendix A

Generalization of Cartesian Combinator Terms

We prove

Theorem 5.34 Given cartesian combinator terms $a, b \in \mathcal{T}_{X \to Y}$, there is no subset of $\mathbf{G}(a,b)$ satisfying Definition 5.9.

Proof Let g be

$$\begin{tikzcd} a & f\langle a, b\rangle \arrow[l, "{[\pi]}"'] \arrow[r, "{[\pi']}"] & b \end{tikzcd}$$

Also, let $\mathbf{G}_g(a,b)$ be the subset of $\mathbf{G}(a,b)$ reachable from g, and let g' be $\langle \theta_1 : t \to a, \theta_2 : t \to b\rangle$. If $\rho : g \to g'$, then $\rho(f)$ must be of the form fs where π appears at least once in s in a string of the form

$$h_{1,1} \ldots h_{1,n_1}\langle h_{2,1} \ldots h_{2,n_2}\langle \cdots \langle h_{m,1} \ldots h_{m,n_m}\pi \cdots\rangle \cdots\rangle\rangle$$

Furthermore, for each $h_{i,j}$, $\theta_1(h_{i,j})$ must either be a projection (if the variable is applied to a pair) or $\mathbf{1}$ (if the variable is applied to another variable). Likewise for π' and θ_2, noting that if $\theta_1(h) = \mathbf{1}$, then $\theta_2(h) = \mathbf{1}$. Finally, let

$$\rho' = \{h \mapsto \mathbf{1} \mid h \mapsto \mathbf{1} \in \theta_1 \cup \theta_2\} \cup \{h \mapsto h\langle \pi', \pi\rangle \mid h \mapsto \pi' \in \theta_1 \text{ or } h \mapsto \pi \in \theta_2\}$$

Then the generalization term of $\rho'\rho(g)$ is of the form

$$f'\langle h_1\langle \cdots h_m\langle a, -\rangle \cdots, -\rangle, h'_1\langle -, \cdots h'_n\langle -, b\rangle \cdots\rangle\rangle$$

where the $-$'s are arbitrary terms, and so $\{f' \mapsto f''\langle f''\langle \mathbf{1}, c!\rangle, f''\langle c!, \mathbf{1}\rangle\rangle$ is always a morphism from $\rho'\rho(g)$ to some g'' in $\mathbf{G}_g(a,b)$. Furthermore, $g'' \not\to g$ since no generalization morphism can reduce the number of occurrences of f'' in g''. Thus there is no subset of $\mathbf{G}_g(a,b)$ (and so of $\mathbf{G}(a,b)$) satisfying the minimality condition of Definition 5.9. $\square$

Since this proof does not depend on duplication, we have actually shown

Corollary A.1 Given *affine* combinator terms $a, b \in \mathcal{T}_{X \to Y}$, $\mathbf{G}(a,b)$ is not well-defined.

where the affine combinators are as defined in Section 5.5.

Appendix B

Results for Relevant Combinator Terms

We show the following:

- matching relevant combinator terms is decidable,
- generalization over relevant combinator terms is well-defined, and
- Algorithm 5.46 is correct.

We assume the notation of Chapter 5. In particular, recall

$$\overline{y_n} = y_1 \cdot (y_2 \cdots (y_{n-1} \cdot y_n))$$

and the definitions of $\mathcal{T}$ (terms), $\mathcal{R}$ (all relevant restructors; 5.36), and $\mathcal{A}$ (associative restructors; 5.44). In addition, we define $\mathcal{AI}$ (associative restructors with insertions) as the set of restructors generated by the production

$$A \ ::= \ \mathbf{1} \mid \alpha \mid \alpha^{-1} \mid \lambda^{-1} \mid \varrho^{-1} \mid AA \mid A \cdot A$$

and $\mathcal{AC}$ (associative-commutative restructors) as the set of restructors generated by the production

$$X \ ::= \ \mathbf{1} \mid \gamma \mid \alpha \mid \alpha^{-1} \mid XX \mid X \cdot X$$

Like $\mathcal{T}$, $\mathcal{R}$, $\mathcal{A}$, $\mathcal{AI}$, and $\mathcal{AC}$ can be indexed:

$$\begin{aligned}
\mathcal{R}_{X \to Y} &= \mathcal{R} \cap \mathcal{T}_{X \to Y} \\
\mathcal{A}_{X \to Y} &= \mathcal{A} \cap \mathcal{T}_{X \to Y} \\
\mathcal{AI}_{X \to Y} &= \mathcal{AI} \cap \mathcal{T}_{X \to Y}
\end{aligned}$$

$$\mathcal{AC}_{X\to Y} \quad = \quad \mathcal{AC} \cap \mathcal{T}_{X\to Y}$$

Furthermore, we define a reduced form for terms such that any occurrences of restructors are at the "leaves" of terms:

Definition B.1 Term $s \in \mathcal{T}_{A\to B}$ is in *restructor-reduced form* if s does not contain any subterms of the form $(p \mapsto q)t$, $\mathbf{1}t$, $t\mathbf{1}$, $t(p \mapsto p)$, or $(p_1 \mapsto q_1)\cdot(p_2 \mapsto q_2)$.

Because the return type of a variable or function cannot be a pair or u,

Lemma B.2 For each $t \in \mathcal{T}_{A\to B}$, there is an $s \in \mathcal{T}_{A\to B}$ in restructor-reduced form such that $s = t$.

The restriction against return types being u ensures that there are no terms of the form $\rho(t \cdot y)$ where $y \in \mathcal{V}_{\mathsf{u}\to\mathsf{u}}$. One use of restructor-reduced form is to define the size of a term:

Definition B.3 The size of a term t in restructor-reduced form is defined as

$$|t| = \begin{cases} 1 + |r| + |s| & \text{if } t = rs \text{ or } t = r \cdot s \\ 1 & \text{otherwise} \end{cases}$$

For a term s not in restructor-reduced form, define $|s|$ by $|s| = |t|$ where t is a term in restructor-reduced form such that $s = t$.

B.1 Matching Relevant Combinator Terms

Because we assume that the result types of variables and constants are not pairs, matching relevant combinator terms is similar to matching second-order λ-terms [HL78]. We present an algorithm and show its correctness.

The following algorithm maintains a set of *disagreement* pairs $\mathcal{D}$ consisting of the terms that have not yet been matched. Each pair in the set is given as $s \overset{?}{\to} t$, where t does not contain any free variables. We assume that all terms s and t are in restructor-reduced form. Initially we have the pair $(\emptyset \,;\, \{s \overset{?}{\to} t\})$, and if there is an appropriate substitution, the algorithm will terminate with $(\theta \,;\, \emptyset)$ such that $\{x \mapsto r \in \theta \mid x \in \mathcal{FV}(s)\} : s \to t$. Note that the algorithm is nondeterministic, so more than one such θ may be returned.

Algorithm B.4

Delete
$$\frac{\theta\,;\,t \xrightarrow{?} t \cup \mathcal{D}}{\theta\,;\,\mathcal{D}}$$

Decompose–Comp
$$\frac{\theta\,;\,\{s_1 s_2 \xrightarrow{?} t_1 t_2\} \cup \mathcal{D}}{\theta\,;\,\{s_1 \xrightarrow{?} t_1,\ s_2 \xrightarrow{?} t_2\} \cup \mathcal{D}}$$

Decompose–Pair
$$\frac{\theta\,;\,\{s_1 \cdot s_2 \xrightarrow{?} t_1 \cdot t_2\} \cup \mathcal{D}}{\theta\,;\,\{s_1 \xrightarrow{?} t_1,\ s_2 \xrightarrow{?} t_2\} \cup \mathcal{D}}$$

Match
$$\frac{\theta\,;\,(xs \xrightarrow{?} t) \cup \mathcal{D} \qquad r \in \mathit{prefix}_x(t)}{\{x \mapsto r\} \circ \theta\,;\,\{x \mapsto r\}(\{(xs, t)\} \cup \mathcal{D})}$$

where *prefix*$_x$ returns all the substitutions for x which may make the diagram

$$\begin{tikzcd}
A \arrow[rr, "t"] \arrow[dr, "s"] & & C \\
& B \arrow[ur, "x"] &
\end{tikzcd}$$

commute. That is,

$$\mathit{prefix}_x(t) =$$
$$\bigcup \begin{cases} \{\tau \mid \tau \in \mathcal{R}_{B \to C}\} & \\ \{K\,\mu'\,\overline{y_m}\,\tau \mid \mu' \in \mathcal{AI}_{X' \to Y},\ m \le n, & \text{if } t = K\,\mu\,\overline{r_n} \text{ for } K \in \mathcal{C}_{Y \to C} \\ \qquad \text{and } \tau \in \mathcal{R}_{B \to X}\} & \qquad \text{and } \mu \in \mathcal{A}_{Y' \to Y} \end{cases}$$

where each y_i is a fresh variable.[1] Note that $K\mu\overline{r_n}$ and $K\mu'\overline{y_m}\tau$ are not in restructor-reduced form. The substitution $x \mapsto K\mu'\overline{y_m}\tau$ is illustrated by the diagram in Figure B.1.

While expensive, prefix$_x$ is computable because

Observation B.5 For any types X and Y, $\mathcal{R}_{X \to Y}$ is finite.

B.1.1 Correctness

We show that Algorithm B.4 is correct; that is, we show

[1] Using the terminology of [HL78], the first substitution corresponds to *projection*, and the second to *imitation*.

$$\begin{tikzcd}
A \arrow[r, "\overline{r_n}"] \arrow[dddrr, "s"] & Y' \arrow[r, "\mu"] & Y \arrow[r, "K"] & C \\
 & & X' \arrow[u, "\mu'"'] & \\
 & & X \arrow[u, "\overline{y_m}"'] & \\
 & & B \arrow[u, "\tau"'] \arrow[uuur, "x"'] &
\end{tikzcd}$$

Figure B.1: Substitution for x when $t = K\mu\overline{r_n}$.

Theorem B.6 (Soundness) Given $s, t \in \mathcal{T}_{X \to Y}$, if Algorithm B.4 halts with $\langle \sigma, \emptyset \rangle$, then $\sigma : s \to t$.

Theorem B.7 (Completeness) Given $s, t \in \mathcal{T}_{X \to Y}$, if $\theta : s \to t$, then there is a sequence of rule applications such that Algorithm B.4 halts with $\langle \theta', \emptyset \rangle$ for $\theta \cong \theta'$.

and

Theorem B.8 (Termination) Algorithm B.4 terminates for all sequences of rule applications.

Soundness follows from the observation that the algorithm maintains the invariant

$$\forall \sigma : \sigma(a_1) = b_1 \ldots \sigma(a_k) = b_k,\ \sigma(\phi(s)) = t$$

where $(\phi, \{a_1 \overset{?}{\to} b_1, \ldots, a_k \overset{?}{\to} b_k\})$ is the result of some sequence of rule applications. Completeness follows from Lemma B.2. Finally, termination follows from a lexicographic ordering based on the sizes of terms, the number of pairs of the form $xt \overset{?}{\to} b$, and the number of free variables. Assume $|S|$ denotes the size of set S, and let

$$\begin{aligned}
\mathcal{S}_b &= \sum_{a \overset{?}{\to} b \,\in\, \mathcal{D}} |b| \\
\mathcal{F} &= |\{xr \overset{?}{\to} b \in \mathcal{D} \mid \text{ x is a free variable}\}| \\
\mathcal{V} &= \sum_{a \overset{?}{\to} b \,\in\, \mathcal{D}} |\mathcal{FV}(a)|
\end{aligned}$$

Then we show termination by the lexicographic ordering on

$$\langle \mathcal{S}_b,\, \mathcal{F},\, \mathcal{V} \rangle$$

Applying *Delete*, *Decompose–Comp*, or *Decompose–Pair* reduces $\mathcal{S}_b$. Applying $x \mapsto K\,\mu\,\overline{y_m}\,\tau$ reduces $\mathcal{F}$. Applying $x \mapsto \tau$ either reduces $\mathcal{F}$ or leaves $\mathcal{F}$ unchanged (such as when $\tau = \mathbf{1}$ and the first symbol of s is a free variable). In either case, applying $x \mapsto \tau$ always reduces $\mathcal{V}$. Thus Algorithm B.4 terminates for all $s, t \in \mathcal{T}_{X \to Y}$.

B.2 Existence of Generalizations

We prove

Theorem 5.38 If $a, b \in \mathbf{T}_{X \to Y}$ are relevant combinator terms, then $\mathrm{MSG}(a, b)$ exists.

In this section, we assume that all terms are in restructor-reduced form. The proof is patterned after 5.16 in that we show the subset of $\mathbf{G}(a, b)$ defined in 5.14, $\mathbf{G}'(a, b)$, satisfies the conditions of Theorem 5.12. The key is the observation

Observation B.9 If a constant or free variable H occurs n times in t and $p \mapsto q$ is in $\mathcal{T}_{B \to C}$ (where $t \in \mathcal{T}_{A \to B}$), then there is an $s = (p \mapsto q)\,t$ such that H occurs n or more times in s.

This is used to show

Lemma B.10 $\mathbf{G}'(a, b)$ is finite.

Proof Let $g_1 = \langle \theta_1 : s \to a, \theta_2 : s \to b \rangle$ be a generalization in $\mathbf{G}'(a, b)$. By Observation B.9, $|a|$ and $|b|$ bound the number of constants that can appear in s. Also, $|a|$ and $|b|$ and the requirement that $\theta_1(x) \neq \theta_2(x)$ bound the number of free variables in s. Finally, the types of the constants and free variables bound the number of terms of the form $p \mapsto q$ in s. Thus $\mathbf{G}'(a, b)$ is finite. $\square$

Next we show that all morphisms to any generalization in $\mathbf{G}'(a, b)$ are unique. Let $\rho, \rho' : g_1 \to g_2$ be in $\mathbf{G}(a, b)$ where g_2 is in $\mathbf{G}'(a, b)$. If $g_2 = \langle \sigma_1 : t \to a, \sigma_2 : t \to b \rangle$, then we must show that $\rho = \rho'$ in the picture

$$\begin{tikzcd}
 & s \arrow[dl, "\theta_1"'] \arrow[d, shift right, "\rho"'] \arrow[d, shift left, "\rho'"] \arrow[dr, "\theta_2"] & \\
a & t \arrow[l, "\sigma_1"'] \arrow[r, "\sigma_2"] & b
\end{tikzcd}$$

Let s be $s[fr]$ for some r where f is the leftmost, outermost variable in s such that $\rho(f) \neq \rho'(f)$. By Observation B.9, there must be a subterm t' in t such that $\rho(fr) = t' = \rho'(fr)$.

First we show

Lemma B.11 $\rho(f) \notin \mathcal{R}$.

Note that this lemma does not hold if the output type of variable can be a pair: the diagram

$$\begin{tikzcd}
& ff' \arrow[dl, "{[\gamma,\mathbf{1}]}"'] \arrow[d, "\rho"] \arrow[dr, "{[\gamma,\mathbf{1}]}"] & \\
\gamma & gh \arrow[l, "{[\gamma,\mathbf{1}]}"'] \arrow[r, "{[\mathbf{1},\gamma]}"] & \gamma
\end{tikzcd}$$

commutes for both $\rho = [\gamma, \gamma gh]$ and $\rho = [gh, \mathbf{1}]$, violating the uniqueness condition of Definition 5.9.

Let $\mathcal{P}^n_X$ be the set of relevant restructors of the form

$$\underbrace{\mathbf{1}_{\mathsf{u}} \cdots \mathbf{1}_{\mathsf{u}}}_{i} \cdot x \cdot \underbrace{\mathbf{1}_{\mathsf{u}} \cdots \mathbf{1}_{\mathsf{u}}}_{j} \mapsto x$$

where $x \in \mathcal{V}_X$ for some base type X and $i + j = n - 1$, and let $\mathcal{P} = \bigcup_{m,Y} \mathcal{P}^m_Y$. Assume f has arity n and output type A. Since A must be a simple type, $\rho(f)$ is a restructor only if $\rho(f) \in \mathcal{P}^n_A$. We show this leads to a contradiction:

Proof of B.11 Assume $\rho(f) = \mu \in \mathcal{P}^n_A$ and $\rho'(f) = \mu_0 p$ for some $\mu_0 \in \mathcal{P}^m_A$. (Note that μ_0 might be $\mathbf{1}_A$.) Since $\sigma_1(\rho(f)) = \mu$, $\sigma_1(\rho'(f)) = \mu$. Every element of $\mathcal{P}$ has an inverse, so there is a μ_0^{-1} such that $\mu = \mu_0\mu_0^{-1}\mu$. Thus $\sigma_1(\rho'(f)) = \sigma_1(\mu_0 p) = \mu = \mu_0\mu_0^{-1}\mu$, or

$$\sigma_1(p) = \mu_0^{-1}\mu$$

Likewise,

$$\sigma_2(p) = \mu_0^{-1}\mu$$

Let x be a variable in p with arity k and output type X. Since $\sigma_1(p) = \sigma_2(p) \in \mathcal{R}$, p must be a term composed entirely of restructors, $\cdot$, and free variables x such that $\sigma_1(x), \sigma_2(x) \in \mathcal{P}^k_X$. But since each restructor in $\mathcal{P}$ has a distinct type, $\sigma_1(x) = \sigma_2(x)$, contradicting $g_2 \in \mathbf{G}'(a, b)$. Therefore, $\rho(f) \notin \mathcal{R}$. $\square$

Since there must be a subterm $t' = \rho(fr) = \rho'(fr)$ in t, Lemma B.11 gives us

Lemma B.12 *head*$(\rho(f)) =$ *head*$(\rho'(f))$

where

$$head(t) = \begin{cases} \mathbf{1} & \text{if } t = \mathbf{1} \\ x & \text{if } t = xs \text{ for a symbol } x \neq \mathbf{1} \end{cases}$$

These results allow us to show Theorem 5.38:

Proof Suppose $\rho(f) = p$ and $\rho'(f) = p'$. Then by Lemma B.12, there are terms p_0, $\overline{q_n}$, and $\overline{q'_n}$ such that $p = p_0\,\overline{q_n}$ and $p' = p_0\,\overline{q'_n}$ for $p_0 \neq \mathbf{1}$ (unless $\rho(f) = \mathbf{1}$) and $\overline{q_n} \neq \overline{q'_n}$. We use induction on the size of substitutions (using a multiset ordering) to show $\overline{q_n} = \overline{q'_n}$ by constructing the commutative diagram

$$\begin{tikzcd}
& s[fr] \arrow[ddl, "\theta_1"'] \arrow[d, "\phi"] \arrow[ddr, "\theta_2"] & \\
& s' \arrow[dl, "\theta'_1"'] \arrow[d, shift right, "\rho_0"'] \arrow[d, shift left, "\rho'_0"] \arrow[dr, "\theta'_2"] & \\
a & t \arrow[l, "\sigma_1"'] \arrow[r, "\sigma_2"'] & b
\end{tikzcd}$$

such that $\rho = \rho_0 \circ \phi$ and $\rho' = \rho'_0 \circ \phi$. Let $\overline{h_n}$ be a collection of fresh variables, and let

$$\begin{aligned}
\theta_1(f) &= p_0\,\overline{q^1_n} \text{ for some collection of terms } \overline{q^1_n} \\
\theta_2(f) &= p_0\,\overline{q^2_n} \text{ for some collection of terms } \overline{q^2_n} \\
\phi &= \{f \mapsto p_0\,\overline{h_n}\} \\
\rho_0 &= \rho \setminus f \cup \{h_1 \mapsto q_1, \ldots, h_n \mapsto q_n\} \\
\rho'_0 &= \rho' \setminus f \cup \{h_1 \mapsto q'_1, \ldots, h_n \mapsto q'_n\} \\
\theta'_1 &= \theta_1 \setminus f \cup \{h_1 \mapsto q^1_1, \ldots, h_n \mapsto q^1_n\} \\
\theta'_2 &= \theta_2 \setminus f \cup \{h_1 \mapsto q^2_1, \ldots, h_n \mapsto q^2_n\}
\end{aligned}$$

Then the diagram commutes for all but $\rho_0 = \rho'_0$, and we can use induction to show $\rho_0(f) = \rho'_0(f)$. Thus $\rho = \rho'$. □

B.3 Correctness of Algorithm 5.46

We conclude Appendix B with a proof that the generalization algorithm for ground relevant combinator terms is correct. The proof is very similar to the proof of correctness for Algorithm 5.27. In this section, assume the steps and relation $\longrightarrow$ are as given by Algorithm 5.46.

Whenever multiple variables are introduced, they are not adjacent, so

Lemma B.13 If g_1 is in $\mathbf{CG}(a,b)$ and $g_1 \longrightarrow g_2$, then g_2 is in $\mathbf{CG}(a,b)$.

Because of the restriction on **Factor-Restructor** that τ not have an inverse,

Lemma B.14 Whenever g_1, g_2 are in $\mathbf{CG}(a,b)$ and $g_1 \longrightarrow g_2$, $g_1 > g_2$.

Finally, we show

Lemma B.15 Whenever $g_r \in \mathrm{MSC}(a,b)$, g_t is in $\mathbf{CG}(a,b)$, and $g_t > g_r$, there is a g_s in $\mathbf{CG}(a,b)$ such that $g_t \longrightarrow g_s$ and $g_s \geq g_r$.

Proof The proof is nearly the same as in Lemma 5.31; we assume the same notation here. One difference is that we must apply the definition of renamings given in Definition 5.42 when selecting $f \in \mathcal{FV}(t)$ such that $\rho_r(f)$ is not a renaming. The other difference is in the analysis of $\rho_r(f)$. Let $\rho_r(f) = Hp$ such that Hp is in restructor-reduced form with H not being a restructor unless $p = \mathbf{1}$. There are three cases for H:

i. H is a restructor μ: $p = \mathbf{1}$, so $\theta_1(f) = \phi_1(\rho_r(f)) = \mu = \phi_2(\rho_r(f)) = \theta_2(f)$, and **Delete** can be applied.

ii. H is a constant function symbol K: $\mathit{head}(\theta_1(f)) = \mathit{head}(\phi_1(\rho_r(f))) = K = \mathit{head}(\phi_2(\rho_r(f))) = \mathit{head}(\theta_2(f))$, so **Factor-Constant** can be applied.

iii. H is a free variable h: By the assumption that $\rho_r(f)$ is not a renaming, $p = \mu\overline{p_n}\mu'$ for some $\mu, \mu' \in \mathcal{AC}$ (the set of associative-commutative restructors) such that for some $p_i \in \overline{p_n}$, p_i has no inverse. There are three cases based on p_i:

   (a) $\mathit{head}(p_i)$ is a free variable h': this would mean that h and h' are adjacent in r, a contradiction.

   (b) p_i is a restructor μ: by assumption, there is no inverse for μ, so **Factor-Restructor** is applicable since μ must occur in both $\theta_1(f)$ and $\theta_2(f)$.

   (c) $\mathit{head}(p_i)$ is a constant K: again, K must occur in both $\theta_1(f)$ and $\theta_2(f)$, so **Factor-Constant** is applicable. □

These results can be used to show soundness (Theorem 5.32) and completeness (Theorem 5.33) as before.

Appendix C

Categorical Foundations of Unification and Generalization

This appendix presents unification and generalization (Section 5.2) in a categorical framework. In particular, we define a category of terms and give categorical definitions of minimally complete sets of unifiers and generalizations. We assume familiarity with the basics of category theory (*cf.* [AM75, AL91, Mac71, Pie91]).

Let $\mathcal{T}$ be a set of "terms" of some kind defined over some countable set of variables $\mathcal{V}$. A substitution is a partial function $\theta : \mathcal{V} \rightarrow \mathcal{T}$ with the domain $dom\,\theta$ defined as the set of variables bound by θ and $ran\,\theta = \{\mathcal{FV}(\theta(x)) \mid x \in dom\,\theta\}$. We say that a substitution θ maps a term r to a term s, written $\theta : r \rightarrow s$, if

- $dom\,\theta = \mathcal{FV}(r)$,
- $ran\,\theta = \mathcal{FV}(s)$, and
- $\theta(r) = s$.

Observe that in most cases $\mathcal{T}$ forms a category with substitutions as arrows. The identity arrows are the substitutions $\theta_{id} : r \rightarrow r$ defined as

$$\theta_{id} = \{x \mapsto x \mid x \in \mathcal{FV}(r)\}$$

Given substitutions $\theta : r \rightarrow s$ and $\sigma : s \rightarrow t$, their composition is defined as

$$\sigma \circ \theta = \{x \mapsto \sigma(\theta(x)) \mid x \in \mathcal{FV}(r)\} : r \rightarrow t$$

Note that the definition and associativity of composition for substitutions depends on $\mathcal{T}$. If composition of substitutions is well-defined and associative, we call $\mathcal{T}$ a *term category.* We use $\mathbf{T}$ to denote an arbitrary term category.

The above treatment is somewhat unusual. First, unlike most examples of categories (particularly in computer science), **T** is not based on sets. However, while this may be rare, there are precedents. Another example of such a category is the category of propositions with proofs as arrows [LS86]. The above treatment is also unusual in that usually terms are viewed as arrows between "types" [Law63]; see [Gog89] for a discussion of unification based on [Law63]. However, the two treatments are closely related; where we write $\theta : r \rightarrow s$, [Gog89] would write

$$\begin{tikzcd}
X \arrow[r, "\theta"] \arrow[dr, "s"] & Y \arrow[d, "r"] \\
 & Z
\end{tikzcd}$$

Likewise, the generalization

$$\begin{tikzcd}
r & g \arrow[l, "\theta_1"'] \arrow[r, "\theta_2"] & s
\end{tikzcd}$$

would be written as

$$\begin{tikzcd}
X_1 \arrow[r, "\theta_1"] \arrow[dr, "r"] & Y \arrow[d, "g"] & X_2 \arrow[l, "\theta_2"'] \arrow[dl, "s"'] \\
 & Z &
\end{tikzcd}$$

and the (weak) unifier

$$\begin{tikzcd}
r \arrow[r, "\theta_1"] & u & s \arrow[l, "\theta_2"']
\end{tikzcd}$$

as

$$\begin{tikzcd}
X_1 \arrow[dr, "r"] & Y \arrow[l, "\theta_1"'] \arrow[r, "\theta_2"] \arrow[d, "u"] & X_2 \arrow[dl, "s"'] \\
 & Z &
\end{tikzcd}$$

Note when terms are viewed as arrows, generalization and unification are not duals of each other.

When **T** is the term category of first-order terms over an empty equational theory, **T** forms a preorder. As noted by Plotkin [Plo70], generalization and unification in this case are defined by products and coproducts, respectively. But in the general case, they are not. For example, in the term category of strings, we have

$$\begin{tikzcd}
 & \mathsf{a} \arrow[dl, "\theta_{id}"'] \arrow[dr, "\theta_{id}"] & \\
\mathsf{a} & xy \arrow[l, "{[\mathsf{a},\epsilon]}"'] \arrow[r, "{[\epsilon,\mathsf{a}]}"] & \mathsf{a}
\end{tikzcd}$$

If the product of a and itself exists, then it must be $g_a = \langle \theta_{id} : \mathsf{a} \to \mathsf{a}, \theta_{id} : \mathsf{a} \to \mathsf{a} \rangle$ because the only substitution defined for a is θ_{id} (where, in this case, $\theta_{id} = \emptyset$). But g_a cannot be the product since any substitution from xy to a must bind either x or y to ϵ. Thus a categorical definition of generalization must allow for multiple maximally specific generalizations. See Example 5.3 for a similar case involving distinct terms. These and other examples lead us to introduce the terms *product set* and *coproduct set* and using these to define generalization and unification. These terms are motivated by the definition of weakly initial sets in [Mac71, Section X.2, Theorem 1]. They are closely related to 'uniquely minimal floorings' in [Gog89] and the dual notion 'uniquely minimal ceilings.'

We first consider generalization. We start by naming the pairs of arrows which appear in the definition of a product and the morphisms between such pairs:

Definition C.1 Given objects a and b in the category $\mathbf{C}$, define a *projection pair* to be a pair of arrows $f : u \to a$ and $f' : u \to b$ for an object u in $\mathbf{C}$. This is written as $\langle f : u \to a, f' : u \to b \rangle$. Given projection pairs $\langle g : v \to a, g' : v \to b \rangle$ and $\langle f : u \to a, f' : u \to b \rangle$, define a *projection morphism* to be an arrow $h : v \to u$ such that the following diagram commutes:

$$\begin{tikzcd}
 & v \arrow[dl, "g"'] \arrow[d, "h"] \arrow[dr, "g'"] & \\
a & u \arrow[l, "f"'] \arrow[r, "f'"] & b
\end{tikzcd}$$

Thus the set of generalizations $\mathbf{G}(a, b)$ for some $a, b \in \mathbf{T}$ forms a category with projection pairs in $\mathbf{T}$ as objects and projection morphisms, also known as *generalization morphisms*, as arrows.

Definition C.2 Given objects a and b in $\mathbf{C}$, a *product set* is a minimal set of projection pairs $P = \{\langle f_i : s_i \to a, f'_i : s_i \to b \rangle\}$ in $\mathbf{C}$ such that for each projection pair q in $\mathbf{C}$, there is a $p \in P$ and a unique projection morphism $h : q \to p$.

This definition is very similar to the definition of maximally specific generalization in 5.9. The only difference is the usage of 'minimal.' In C.2, 'minimal' is used to constrain the product set to contain as few projection pairs as possible. In 5.9, 'minimal' constrains MSG so that if

$h : g \to g'$ for $g, g' \in \mathrm{MSG}(a, b)$, $h = \theta_{id}$. The following lemma, adapted from [Gog89], shows that these two usages are equivalent:

Lemma C.3 Let P be a set of projection pairs over a and b in $\mathbf{C}$ such that for each projection pair p for a and b in $\mathbf{C}$, there is a $p' \in P$ such that $p \to p'$. Then P is minimal (in the sense of containing as few elements as possible) if and only if for each $q, q' \in P$, $q \to q'$ implies $q = q'$.

Proof Suppose $q, q' \in P$ and $q \to q'$. Then by the assumption that P is minimal, $q = q'$. Conversely, suppose P is not minimal; then there are $q, q' \in P$ such that $q \to q'$ for $q \neq q'$, a contradiction. $\square$

Thus MSG is an instance of product sets. Furthermore, Lemma C.3 means that product sets are unique up to an isomorphism:

Theorem C.4 If P_1 and P_2 are product sets in $\mathbf{C}$, $P_1 \cong P_2$.

The proof is constructed by making minor modifications to the proof of Theorem 5.7:

Proof By assumption, for each $p_1 \in P_1$ there is a $p_2 \in P_2$ and a unique morphism $h : p_1 \to p_2$. Likewise, there is a $q_1 \in P_1$ and a unique morphism $h' : p_2 \to q_1$. But because morphisms compose and P_1 satisfies the minimality condition, $p_1 = q_1$. Furthermore, $h' \circ h = \theta_{id} : p_1 \to p_1$ and $h \circ h' = \theta_{id}$, giving $P_1 \cong P_2$. $\square$

Thus product sets are unique and can be used to define generalization.

We use the dual notions to define unification:

Definition C.5 Given objects a and b in the category $\mathbf{C}$, define an *injection pair* to be a pair of arrows $f : a \to u$ and $f' : b \to u$ for an object u in $\mathbf{C}$. This is written as $\langle f : a \to u, f' : b \to u\rangle$. Given injection pairs $\langle g : a \to v, g' : b \to v\rangle$ and $\langle f : a \to u, f' : b \to u\rangle$, define an *injection morphism* to be an $h : v \to u$ such that the following diagram commutes:

$$\begin{tikzcd}
a \arrow[r, "g"] \arrow[dr, "f"'] & v \arrow[d, "h"] & b \arrow[l, "g'"'] \arrow[dl, "f'"] \\
& u &
\end{tikzcd}$$

Thus the set of unifiers of a and b forms a category with injection pairs as objects and injection morphisms as arrows.

Definition C.6 Given objects a and b in $\mathbf{C}$, a *coproduct set* is a minimal set of injection pairs $I = \{\langle f_i : a \to s_i, f'_i : b \to s_i\rangle\}$ in $\mathbf{C}$ such that for each injection pair q In $\mathbf{C}$, there is a $p \in I$ and a unique injection morphism $h : p \to q$.

That is, if product sets parallel limits, coproduct sets parallel colimits. Again, this definition is similar to the definition of uniquely minimal complete sets of unifiers (5.5 with uniqueness) except for the usage of 'minimal.' The dual of Lemma C.3 proves that the two usages are equivalent:

Lemma C.7 Let I be a set of injection pairs over a and b in $\mathbf{C}$ such that for each injection pair p for a and b in $\mathbf{C}$, there is a $p' \in I$ such that $p' \rightarrow p$. Then I is minimal (in the sense of containing as few elements as possible) if and only if for each $q, q' \in I$, $q' \rightarrow q$ implies $q = q'$.

Thus uniquely minimal complete sets of unifiers are instances of coproduct sets. As with product sets, Lemma C.7 implies that coproduct sets are unique up to an isomorphism:

Theorem C.8 If I_1 and I_2 are coproduct sets in $\mathbf{C}$, $I_1 \cong I_2$.

Proof Dual of Theorem C.4. □

Thus coproduct sets are unique and can be used to define unification. Investigating unification in this framework is left as future work.

The categorical treatment of generalization and unification contributes by motivating the condition that morphisms to product sets (or from coproduct sets) be unique. As discussed in Section 5.2, minimally complete sets of generalizations and unifiers are not necessarily canonical. For applications in which canonical sets of generalizations and unifiers are useful, the uniqueness condition gives sets which are canonical up to an isomorphism. While there are other ways to define canonical sets, the uniqueness condition has the advantage of being consistent with the standard practice in category theory of using unique morphisms to define special objects.

Thus we have defined generalization and unification using concepts from category theory and have shown that the definitions are equivalent to those in Section 5.2. This helps clarify the relationship between generalization and unification. It also helps motivate our definitions for generalization morphisms and maximally specific generalizations.

Bibliography

[AL91] Andrea Asperti and Giuseppe Longo. *Categories, Types, and Structures: An Introduction to Category Theory for the Working Computer Scientist.* MIT Press, Cambridge, MA, 1991.

[AM75] Michael A. Arbib and Ernest G. Manes. *Arrows, Structures, and Functors: The Categorical Imperative.* Associated Press, New York, 1975.

[AMCP84] P.B. Andrews, D. Miller, E. Cohen, and F. Pfenning. Automating higher-order logic. *Contemporary Mathematics*, 29:169–192, 1984.

[ASS85] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programms.* MIT Press, Cambridge, Mass., 1985.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA, 1986.

[Baa91] Fanz Baader. Unification, weak unification, upper bound, lower bound, and generalization problems. In Ronald V. Book, editor, *4th Int. Conf. on Rewriting Techniques and Applications*, volume 488 of *LNCS*, pages 86–97, Como, Italy, April 1991. Springer-Verlag.

[Bal85] Robert Balzer. A 15 year perspective on automatic programming. *IEEE Transactions on Software Engineering*, SE-11(11):1257–1268, November 1985.

[Bau79] Michael A. Bauer. Programming by examples. *Artificial Intelligence*, 12:1–21, 1979.

[Bax90] Ira Baxter. *Transformational Maintenance by Reuse of Design Histories.* PhD thesis, University of California, Irvine, USA, November 1990.

[Bax92] Ira Baxter. Design maintenance systems. *CACM*, 35(4):73–89, April 1992.

[BCG83] Robert Balzer, Thomas E. Cheatham, Jr., and Cordell Green. Software technology in the 1990's: Using a new paradigm. *IEEE Computer*, 16:39–45, November 1983.

[BCST93] R. F. Blute, J. R. B. Cockett, R. A. G. Seely, and T. H. Trimble. Natural deduction and coherence for weakly distributive categories. Available by anonymous `ftp` from `triples.math.mcgill.ca`, March 1993.

[BD77] R. M. Burstall and John Darlington. A transformation system for developing recursive programs. *Journal of the ACM*, 24(1):44–67, January 1977.

[Bel90] Françoise Bellegarde. A matching process modulo a theory of categorical products. In H. Kirchner and W. Wechler, editors, *Workshop on Algebraic and Logic Programming*, volume 463 of *LNCS*, pages 270–282, 1990.

[BGW76] Robert Balzer, Neil Goldman, and David Wile. On the transformational implementation approach to programming. In *Proc. 2nd Int. Conf. on Software Engineering*, pages 337–344, San Francisco, Calif., October 1976.

[Bha91] Sanjay Bhansali. *Domain-Based Program Synthesis Using Planning and Derivational Analogy.* PhD thesis, University of Illinois, Urbana, IL, May 1991.

[BRH94] Francois Bronsard, Uday S. Reddy, and Robert W. Hasker. Induction using term orderings. In Bundy [Bun94], pages 102–117.

[BS93] Franz Baader and Jörg H. Siekmann. Unification theory. In D. M. Gabbay, C. J. Hogger, and J. A. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming.* Oxford University Press, 1993.

[Bun94] Alan Bundy, editor. *12th Int. Conf. on Automated Deduction*, volume 814 of *Lecture Notes in Artificial Intelligence*, Nancy, France, 1994. Springer-Verlag.

[Car83] Jaime G. Carbonell. Learning by analogy: Formulating and generalizing plans from past experience. In Michalski et al. [MCM83], chapter 5, pages 137–161.

[Car86] Jaime G. Carbonell. Derivational analogy: A theory of reconstructive problem solving and expertise acquisition. In Michalski et al. [MCM86], chapter 14, pages 371–392.

[Chu40] A. Church. A formulation of the simple theory of types. *Journal of Symbolic Logic*, 5:56–58, 1940.

[CLR90] Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. *Introduction to Algorithms.* MIT Press, Cambridge, MA, 1990.

[Coh88] Avra Cohn. A proof of correctness of the Viper microprocessor: The first level. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, chapter 2, pages 27–71. Kluwar Academic Pub., Boston, 1988.

[Con86] R. L. Constable, *et. al. Implementing Mathematics with the Nuprl Proof Development System.* Prentice-Hall, New York, 1986.

[Coo90a] Diane J. Cook. Application of analogical planning to engineering design. In *6th Annual Conference on Artificial Intelligence Applications*, pages 244–249, Santa Barbara, CA, May 1990.

[Coo90b] Diane J. Cook. *Base Selection in Analogical Planning.* PhD thesis, University of Illinois, Urbana, IL, 1990.

[Cur93] P.-L. Curien. *Categorical Combinators, Sequential Algorithms and Functional Programming.* Birkhauser, Boston, second edition, 1993.

[CW85] Luca Cardelli and Peter Wegner. On understanding types, data abstraction, and polymorphism. *ACM Computing Surveys*, 17(4):471–522, December 1985.

[Dar71] J. L. Darlington. A partial mechanization of second-order logic. In *Machine Intelligence*, volume 6, chapter 7, pages 91–100. American Elsevier, New York, 1971.

[Dar81] John Darlington. The structured description of algorithm derivations. In J.W. De Bakker and J.C. Van Vliet, editors, *Proceedings of the International Symposium on Algorithmic Languages*, pages 221–250, Amsterdam, 1981. North-Holland Publishing Co.

[Der75] Nachum Dershowitz. A simplified loop-free algorithm for generating permutations. *BIT*, 15:158–164, 1975.

[Der87] Nachum Dershowitz. Termination of rewriting. *Journal of Symbolic Computation*, 3:69–116, 1987.

[DJ90] Nachum Dershowitz and Jean-Pierre Jouannaud. Rewrite systems. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science, Vol. B: Formal Methods and Semantics*, chapter 6, pages 243–320. North-Holland, Amsterdam, 1990.

[DM77] N. Dershowitz and Z. Manna. The evolution of programs: Automatic program modification. *IEEE Transactions on Software Engineering*, SE-3(6):377–385, November 1977.

[DM86] Gerald DeJong and Raymond Mooney. Explanation-based learning: An alternative view. *Machine Learning*, 1:145–176, 1986.

[DMLP79] Richard A. De Millo, Richard J. Lipton, and Alan J. Perlis. Social processes and proofs of theorems and programs. *CACM*, 22(5):271–280, May 1979.

[Dow92] Gilles Dowek. Third order matching is decidable. In *Proceedings, Seventh Annual IEEE Symposium on Logic in Computer Science*, Santa Cruz, California, June 1992. IEEE Computer Society Press.

[Ede85] Elmar Eder. Properties of substitutions and unifications. *Journal of Symbolic Computation*, 1:31–46, 1985.

[Eva68] Thomas G. Evans. A program for the solution of a class of geometric-analogy intelligence-test questions. In Marvin Minsky, editor, *Semantic Information Processing*, chapter 5, pages 271–353. MIT Press, 1968.

[Fal88] Brian Falkenhainer. *Learning from Physical Analogies: A Study in Analogy and the Explanation Process.* PhD thesis, University of Illinois, Urbana, IL, December 1988.

[Far88] William M. Farmer. A unification algorithm for second-order monadic terms. *Pure and Applied Logic*, 39:131–174, 1988.

[Fea79] Martin S. Feather. *A System for Developing Programs by Transformation.* PhD thesis, Univ. of Edinburgh, 1979.

[Fea82] Martin S. Feather. A system for assisting program transformation. *ACM TOPLAS*, 4(1):1–20, January 1982.

[FFG86] Brian Falkenhainer, Kenneth D. Forbus, and Dedre Gentner. The structure–mapping engine. In *Proceedings of the American Association for Artificial Intelligence*, pages 272–277, August 1986.

[FFG89] Brian Falkenhainer, Kenneth D. Forbus, and Dedre Gentner. The structure–mapping engine: Algorithm and examples. *Artificial Intelligence*, 41(1):1–63, November 1989.

[FH94] Amy Felty and Douglas Howe. Generalization and reuse of tactic proofs. Submitted to LAPR '94, 1994.

[FHN72] Richard E. Fikes, Peter E. Hart, and Nils J. Nilsson. Learning and executing generalized robot plans. *Artificial Intelligence*, 4(3):189–208, 1972.

[Fic85] Stephen F. Fickas. Automating the transformational development of software. *IEEE Transactions on Software Engineering*, SE-11(11):1268–1277, November 1985.

[FM88] Amy Felty and Dale A. Miller. Specifying theorem provers in a higher-order logic programming language. In Ewing Lusk and Ross Overbeek, editors, *9th International Conference on Automated Deduction*, volume 310 of *LNCS*, pages 61–80, Berlin, May 1988. Springer-Verlag.

[FP90] Alan M. Frisch and C. David Page Jr. Generalization with taxonomic information. In *Proceedings of the American Association for Artificial Intelligence*, pages 777–761, Boston, MA, 1990.

[Gen83] Dedre Gentner. Structure-mapping: A theoretical framework for analogy. *Cognitive Science*, 7:155–170, 1983.

[Gen89] Dedre Gentner. The mechanisms of analogical learning. In Stella Vosriadou and Andrew Ortony, editors, *Similarity and Analogical Reasoning*, chapter 7, pages 199–241. Cambridge University Press, Cambridge, 1989.

[GJCOG89] Mike Gordon, Jeff Joyce, Rachel Cardell-Oliver, and Elsa Gunter. The HOL system TUTORIAL. Unpublished, December 1989.

[Gog89] Joseph A. Goguen. What is unification? A categorical view of substitution, equation, and solution. In Hassan Aït-Kaci and Maurice Nivat, editors, *Resolution of Equations in Algebraic Structures*, volume 1, chapter 6, pages 217–261. Academic Press, 1989.

[Gol90] Allen Goldberg. Reusing software developments. *SIGSOFT Software Engineering Notes*, 15(6):107–119, December 1990. Proceedings, Fourth ACM SIGSOFT Symposium on Software Development Environments.

[Gor88] Michael J.C. Gordon. HOL: A proof generating system for higher-order logic. In Graham Birtwistle and P. A. Subrahmanyam, editors, *VLSI Specification, Verification and Synthesis*, chapter 3, pages 73–128. Kluwar Academic Pub., Boston, 1988.

[GR83] Adele Goldberg and David Robson. *Smalltalk-80: The Language and Its Implementation.* Addison-Wesley, Reading, Mass., 1983.

[Gri81] Sam Grier. A tool that detects plagiarism in Pascal programs. *ACM SIGCSE Bulletin*, 13(1):15–20, February 1981.

[HA88] Michael N. Huhns and Ramón D. Acosta. Argo: A system for design by analogy. *IEEE Expert*, 3(3):53–68, 1988.

[Hag89] Masami Hagiya. Generalization from partial parameterization in higher-order type theory. *TCS*, 53:113–139, 1989.

[Hag90] Masami Hagiya. Programming by example and proving by example using higher-order unification. In M. E. Stickel, editor, *10th Int. Conf. on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 588–602, Kaiserslautern, FRG, July 1990. Springer-Verlag.

[Hag91a] Masami Hagiya. From programming-by-example to proving-by-example. In T. Ito and A. R. Meyer, editors, *TACS*, pages 397–420, September 1991.

[Hag91b] Masami Hagiya. Synthesis of rewrite programs by higher-order and semantic unification. *New Generation Computing*, 8:403–420, 1991.

[Hal89] Rogers P. Hall. Computational approaches to analogical reasoning: A comparative analysis. *Artificial Intelligence*, 39:39–120, 1989.

[Ham88] Dave Hammerslag. Treemacs manual. Technical Report UIUCDCS-R-88-1427, University of Illinois at Urbana-Champaign, May 1988.

[HJ92] Juergen Haas and Bharat Jayaraman. From contrext-free to definite-clause grammars (generalization from examples). In Miller [Mil92], pages 113–122.

[HL78] Gérard Huet and Bernard Lang. Proving and applying program transformations expressed with second-order patterns. *Acta Informatica*, 11(55):31–55, 1978.

[HM88] John Hannan and Dale Miller. Uses of higher-order unification for implementing program transformers. In R. A. Kowalski and K. A. Bowen, editors, *Logic Programming: Proceedings of the Fifth International Conference and Symposium*, volume 2, pages 942–959, Seattle, August 1988. MIT Press.

[HR92] Robert W. Hasker and Uday S. Reddy. Generalization at higher types. In Miller [Mil92], pages 123–139.

[Hue75] Gérard Huet. A unification algorithm for typed λ-calculus. *Theoretical Computer Science*, 1:27–57, 1975.

[Hue76] Gérard Huet. Résolution d'équations dans des langages d'order $1, 2, \ldots, \omega$ (these d'etat), December 1976.

[Hue94] Gerard Huet. Personal communication, August 1994. Reference to proof by Vincent Padovani.

[IEE91] IEEE Computer Society Press. *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science*, Amsterdam, The Netherlands, July 1991.

[Jac92] Bart Jacobs. Semantics of weakening and contraction. Manuscript, Cambridge University, November 1992. To appear in Annals of Pure and Applied Logic.

[Jon92] Doug Jones. Personal communication, March 1992.

[KB70] Donald E. Knuth and Peter B. Bendix. Simple word problems in universal algebras. In John Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–297, Oxford, 1970. Pergamon Press.

[KK71] R. A. Kowalski and D. Kuehner. Linear resolution with selection function. *Artificial Intelligence*, 2:227–260, 1971.

[KL80] Sam Kamin and Jean-Jacques Lévy. Attempts for generalising the recursive path orderings. Unpublished note, Dept. of Computer Science, University of Illinois, Urbana, IL, February 1980.

[Kli71] R. E. Kling. A paradigm for reasoning by analogy. *Artificial Intelligence*, 2(2):147–178, 1971.

[KM71] G. M. Kelly and S. MacLane. Coherence in closed categories. *Journal of Pure and Applied Algebra*, 1(1):97–140, January 1971.

[Knu74] D. K. Knuth. Structured programming with `go to` statements. *Computer Surveys*, 6(4):261–301, December 1974.

[Kow79] R.A. Kowalski. Algorithm = logic + control. *Communications of the ACM*, 22(7):424–436, July 1979.

[KW94] Thomas Kolbe and Christoph Walther. Reusing proofs. In A. Cohn, editor, *11th European Conference on Artificial Intelligence*, pages 80–84. John Wiley & Sons, Ltd., 1994.

[Lam87] J. Lambek. Multicategories revisited. In John W. Gray and Andre Scedrov, editors, *Categories in Computer Science and Logic*, Boulder, Colorado, June 1987. American Mathematical Society. Published as Vol. 92 of Contemporary Mathematics, 1989.

[Law63] F. William Lawvere. Functional semantics of algebraic theories. In *National Academy of Sciences*, 1963.

[Lin88] Peter A. Lindsay. A survey of mechanical support for formal reasoning. *Software Engineering Journal*, 3(1):3–27, January 1988.

[LLtG90] Bil Lewis, Dan LaLiberte, and the GNU Manual Group. *GNU Emacs Lisp Reference Manual, Edition 1.03.* Free Software Foundation, Cambridge, MA, December 1990.

[LMM88] J-L. Lassez, M. J. Maher, and K. Marriott. Unification revisited. In Jack Minker, editor, *Foundations of Deductive Databases and Logic Programming*, chapter 15, pages 587–625. Morgan Kauffmann Publishers, 1988.

[LRN86] John E. Laird, Paul S. Rosenbloom, and Allen Newell. Chunking in Soar: The anatomy of a general learning mechanism. *Machine Learning*, 1:11–46, 1986.

[LS86] J. Lambek and P. J. Scott. *Introduction to higher order categorical logic*, volume 7 of *Cambridge Studies in Advanced Mathematics.* Cambridge University Press, 1986.

[LS92] Janusz Laski and Wojciech Szermer. Identification of program modifications and its application to software maintenance. In Marc Kellner, editor, *Conference on Software Maintenance*, pages 282–290, Orlando, Florida, November 1992. IEEE.

[Mac71] Saunders MacLane. *Categories for the Working Mathematician*, volume 5 of *Graduate Texts in Mathematics.* Springer-Verlag, New York, 1971.

[MB87] Jack Mostow and Mike Barley. Automated reuse of design plans. In W. E. Eder, editor, *Proceedings of the International Conference on Engineering Design*, pages 632–647, August 1987.

[MCA82] Dale A. Miller, Eve Longini Cohen, and Peter B. Andrews. A look at TPS. In *6th Conf. on Automated Deduction*, volume 130 of *LNCS*, pages 50–69, NY, June 1982. Springer-Verlag.

[McD79] John McDermott. Learning to use analogies. In *6th International Joint Conference on Artificial Intelligence*, volume 1, pages 568–576, 1979.

[MCM83] Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors. *Machine Learning: An Artificial Intelligence Approach.* Tioga Publishing Company, Los Altos, CA, 1983.

[MCM86] Ryszard S. Michalski, Jaime G. Carbonell, and Tom M. Mitchell, editors. *Machine Learning: An Artificial Intelligence Approach*, volume II. Morgan Kaufmann, Los Altos, CA, 1986.

[MF89] Jack Mostow and Greg Fisher. Replaying transformational derrivations of heuristic search algorithms in DIOGENES. In *Workshop on Case-Based Reasoning*, pages 94–99, San Mateo, CA, June 1989. DARPA, Morgan Kaufmann Publishers, Inc.

[MH91] Kanth Miriyala and Mehdi T. Harandi. Automatic derivation of formal software specifications from informal descriptions. *IEEE Transactions on Software Engineering*, 17(10):1126–1142, October 1991.

[Mic86] Ryszard S. Michalski. Understanding the nature of learning: Issues and research directions. In Michalski et al. [MCM86], chapter 1, pages 3–25.

[Mil78] Robin Milner. A theory of type polymorphism in programming. *Journal of Computer and System Sciences*, 17(3):348–375, December 1978.

[Mil91] Dale Miller. Unification of simply typed lambda-terms as logic programming. In P.K. Furukawa, editor, *Proc. Joint Int. Conf. on Logic Programming*, pages 253–281, 1991.

[Mil92] Dale Miller, editor. *Proceedings of the Workshop on the λProlog Programming Language*, Philadelphia, PA, USA, 1992. Available as Univ. of Pennsylvania tech. report MS-CIS-92-86.

[MKKC86] T. M. Mitchell, R. M. Keller, and S. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1(1):47–80, January 1986.

[MM85] W. Miller and E. Myers. A file comparison program. *Software Practice and Experience*, 15(11):1025–1040, 1985.

[MMS85] Tom M. Mitchell, Sridhar Mahadevan, and Louis I. Steinberg. LEAP: A learning apprentice for VLSI design. In *9th Intern. Joint Conf. on Artificial Intelligence*, pages 573–580, Los Angeles, CA, 1985.

[MN86] Dale Miller and Gopalan Nadathur. Higher-order logic programming. In *Third Intern. Logic Program. Conf.*, volume 225 of *LNCS*, pages 448–462, London, June 1986. Springer-Verlag.

[MN87] Dale Miller and Gopalan Nadathur. A logic programming approach to manipulating formulas and programs. In *IEEE Symposium on Logic Programming*, San Francisco, September 1987.

[Mos89] Jack Mostow. Design by derivational analogy: Issues in the automated replay of design plans. *Artificial Intelligence*, 40:119–184, 1989.

[Mye86] Eugene Myers. An $O(ND)$ difference algorithm and its variations. *Algorithmica*, 1(2):251–266, 1986.

[Nip91] Tobias Nipkow. Higher-order critical pairs. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science* [IEE91], pages 74–85.

[NP92] Gopalan Nadathur and Frank Pfenning. Types in higher-order logic programming. In Frank Pfenning, editor, *Types in Logic Programming*, chapter 9, pages 245–283. MIT Press, Cambridge, Mass., 1992.

[NTFT91] Fujio Nishida, Shinobu Takamatsu, Yoneharu Fujita, and Tadaaki Tani. Semi-automatic program construction from specifications using library modules. *IEEE Transactions on Software Engineering*, 17(9):853–871, September 1991.

[Owe90] Stephen Owen. *Analogy for Automated Reasoning*. Academic Press, Boston, 1990.

[Pag93] Charles David Page Jr. *Anti-Unification in Constraint Logics: Foundations and Applications to Learnability in First-Order Logic, to Speed-Up Learning, and to Deduction*. PhD thesis, University of Illinois, Urbana, IL, July 1993.

[Pau86] Lawrence C. Paulson. Natural deduction as higher-order resolution. *Journal of Logic Programming*, 3:237–258, 1986.

[PE88] Frank Pfenning and Conal Elliott. Higher-order abstract syntax. In *Conference on Programming Language Design and Implementation*, pages 199–208. ACM SIGPLAN, June 1988. Published in *SIGPLAN Notices*, Vol. 23, Number 7, July 1988.

[PF92] C. David Page Jr. and Alan M. Frisch. Generalization and learnability: A study of constrained atoms. In S. H. Muggleton, editor, *Inductive Logic Programming*, pages 29–61. Academic Press, London, 1992.

[Pfe88] Frank Pfenning. Partial polymorphic type inference and higher-order unification. In *ACM Conference on Lisp and Functional Programming*, pages 153–163, Snowbird, Utah, July 1988.

[Pfe91] Frank Pfenning. Unification and anti-unification in the calculus of constructions. In *Proceedings, Sixth Annual IEEE Symposium on Logic in Computer Science* [IEE91], pages 74–85.

[Pie73] T. Pietrzykowski. A complete mechanization of second-order type theory. *Journal of the ACM*, 20(2):333–365, April 1973.

[Pie91] Benjamin C. Pierce. *Basic Category Theory for Computer Scientists.* MIT Press, Cambridge, MA, 1991.

[PJ72] T. Pietrzykowski and D.C. Jensen. A complete mechanization of (ω)-order type theory. In *Proc. ACM Conference*, pages 82–93, 1972.

[Pla80] David A. Plaisted. Abstraction mappings in mechanical theorem proving. In W. Bibel and R. Kowalski, editors, *5th International Conference on Automated Deduction*, volume 87 of *LNCS*, pages 264–280, Les Arcs, France, 1980. Springer-Verlag.

[Plo70] Gordon D. Plotkin. A note on inductive generalization. In *Machine Intelligence*, volume 5, chapter 8, pages 153–163. American Elsevier, New York, 1970.

[Plo71] Gordon D. Plotkin. A further note on inductive generalization. In *Machine Intelligence*, volume 6, chapter 8, pages 101–124. American Elsevier, New York, 1971.

[Plo72] Gordon D. Plotkin. Building-in equational theories. In *Machine Intelligence*, volume 7, chapter 4, pages 73–90. American Elsevier, New York, 1972.

[Pol57] G. Polya. *How To Solve It.* Doubleday, Garden City, NY, second edition, 1957.

[Pre94] Christian Prehofer. Decidable higher-order unification problems. In Bundy [Bun94], pages 635–649.

[Rea88] Stephen Read. *Relevant Logic: A Philosophical Examination of Inference.* Basil Blackwell, Inc., New York, NY, 1988.

[Red88a] U. S. Reddy. Transformational derivation of programs using the Focus system. *SIGSOFT Software Engineering Notes*, 13(5):163–172, November 1988. Proceedings, ACM SIGSOFT/SIGPLAN Third Software Engineering Symposium on Practical Software Development Environments; also published as *SIGPLAN Notices*, Feb. 1989.

[Red88b] Uday S. Reddy. Transforming sequential programs to parallel ones. Preprint, University of Illinois at Urbana-Champaign, January 1988.

[Red90a] Uday S. Reddy. Formal methods in transformational derivation of programs. *Software Engineering Notices*, 15(4):104–114, September 1990. Proceedings of the ACM SIGSOFT Workshop on Formal Methods in Software Development.

[Red90b] Uday S. Reddy. Term rewriting induction. In M. Stickel, editor, *10th International Conference on Automated Deduction*, volume 449 of *Lecture Notes in Artificial Intelligence*, pages 162–177. Springer-Verlag, 1990.

[Rey70] John C. Reynolds. Transformational systems and the algebraic structure of atomic formulas. In *Machine Intelligence*, volume 5, chapter 7, pages 135–151. Edinburgh Univ. Press, Edinburgh, 1970.

[RND77] Edward M. Reingold, Jurg Nievergelt, and Narsingh Deo. *Combinatorial Algorithms: Theory and Practice.* Prentice Hall, Inc., New Jersey, 1977.

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution princple. *Journal of the ACM*, 12(1):23–41, April 1965.

[Rob69] J. A. Robinson. Mechanizing higher-order logic. In Bernard Meltzer and Donald Michie, editors, *Machine Intelligence*, volume 4, chapter 9, pages 151–169. American Elsevier, New York, 1969.

[Roe92] Ken Roe. Automating program derivation in a transformational environment. Master's thesis, University of Illinois at Urbana-Champaign, 1992.

[RW88] Charles Rich and Richard C. Waters. Automated programming: Myths and prospects. *IEEE Computer*, pages 40–51, August 1988.

[Sil86] Bernard Silver. *Meta-level Inference.* Elsevier Science Publishers, New York, NY, 1986.

[SM85] Louis I. Steinberg and Tom M. Mitchell. The redesign system: A knowledge-based approach to VLSI CAD. *IEEE Design and Test of Computers*, 2(1):45–54, February 1985.

[SS83] William L. Scherlis and Dana S. Scott. First steps towards inferential programming. In R. E. A. Mason, editor, *Information Processing*, pages 199–212, North-Holland, 1983. Elsevier Science Publishers.

[Ste87] Louis I. Steinberg. Design as refinement plus constraint propagation: The VEXED experience. In *6th National Conference on Artificial Intelligence*, volume 2, pages 830–835. AAAI, July 1987.

[Ste89] Joachim Steinbach. Extensions and comparison of simplification orderings. In N. Dershowitz, editor, *3rd Int. Conf. on Rewriting Techniques and Applications*, volume 355 of *LNCS*, pages 434–448, Chapel Hill, North Carolina, April 1989. Springer-Verlag.

[Str91] Bjarne Stroustrup. *The C++ Programming Language*. Addison-Wesley Publishing Company, Reading, Mass., second edition, 1991.

[Vel92] Manuela M. Veloso. *Learning by Analogical Reasoning in General Problem Solving.* PhD thesis, Carnegie Mellon University, Pittsburgh, Penn., August 1992.

[Wil83] David S. Wile. Program developments: Formal explanations of implemenations. *Communications of the ACM*, 26(11):902–911, November 1983.

[Wol93] D. A. Wolfram. *The Clausal Theory of Types.* Cambridge Univ. Press, 1993.

Vita

Robert W. Hasker was born on March 25, 1963, in Woodland, California. He recieved a Bachelor of Science in Mathematics from Wheaton College, Illinois, in December of 1984. After graduation, he worked for Software Consulting Specialists in Fort Wayne, Indiana, where he participated in a number of projects including leading a study on the automated programming of aircraft displays. He entered graduate school at the Urbana-Champaign campus of the University of Illinois in the fall of 1986, where he held both teaching and research assistantships. In the 1993-94 school year, he held a position as visiting lecturer at the University of Illinois.